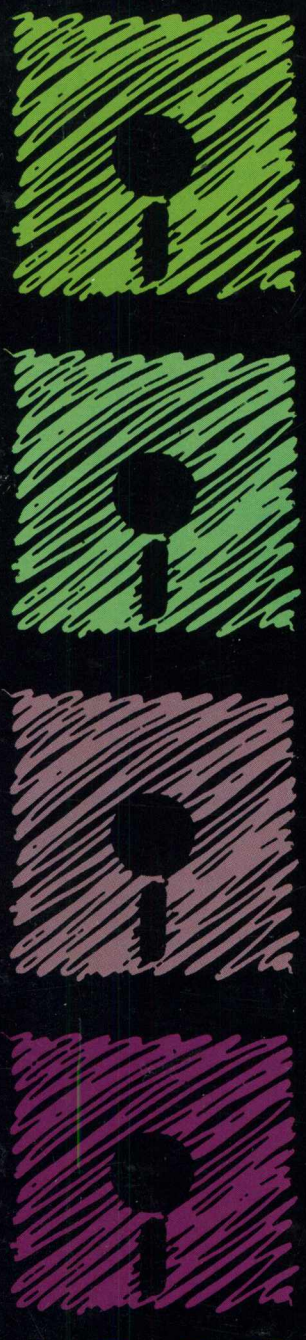
Dr. Dobb's Journal



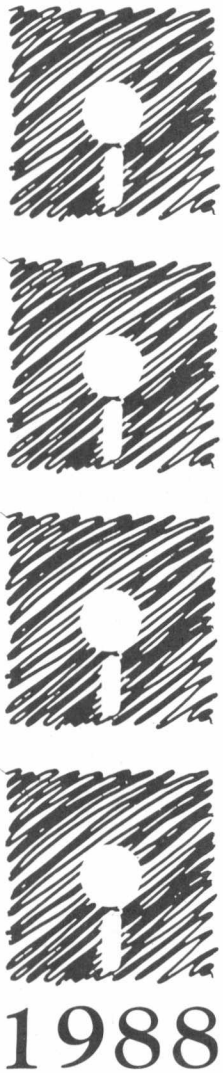
VOLUME

13

Software Tools

1988

Dr. Dobb's Journal



VOLUME

13

Software Tools

1988

A Publication of M&T Publishing, Redwood City, CA

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

M&T Books
General Manager, Ellen Ablow
Editorial Project Manager, Michelle Hudun
Editor and Indexer, Sally J. Brenton
Cov er Art Director, Michael Hollister
Cover Designer, Barbara Mautz
Production Artists, Beth Auten, Lynn Sanford

© 1989 by M&T Publishing, Inc.

Printed in the United States of America
First Edition published 1989

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

ISBN 1-55851-027-3

$39.95

92 91 90 89 4 3 2 1

About M&T Publishing and *Dr. Dobb's Journal*

M&T Publishing, Inc., was founded in 1981 as the American subsidiary of Markt & Technik A.G., the leading West German publisher of computer magazines, books, and software. In addition to its active role in the international microcomputer marketplace, M&T publishes three monthly magazines: *Dr. Dobb's Journal of Software Tools*, *DBMS*, and *LAN Technology*. M&T also publishes and a quarterly magazine, *PC Accounting*. In addition, M&T publishes microcomputer books and software under the imprint M&T Books.

Since its first issue in 1976, *Dr. Dobb's Journal* has maintained a unique voice among microcomputer publications. The first issue was published to provide short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it immediately went into monthly production. *DDJ* has led the way since then by focusing on important advances in microcomputers, printing public-domain software, and fostering vital reader interaction on technical matters.

Today, *Dr. Dobb's Journal* remains the primary source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the editorial content of *DDJ* has ever gone out of print. Volumes One through Thirteen of this series contain the entire editorial content of *DDJ* 's first thirteen years and are available through M&T Books.

Through *Dr. Dobb's Journal* and its other software publications, M&T Publishing is providing microcomputer professionals and enthusiasts with the most advanced and useful software information available.

Editor's Preface

The simple fact that you're holding this book in your hands means that *Dr. Dobb's Journal* has achieved one of its most important goals—to share information that has lasting reference value. Unlike most other computer magazines, *DDJ* has been from the very start—and 1988 is our thirteenth year—dedicated to providing information and programming techniques that will be just as useful two or three (or more) years down the road as they were when they first appeared on the magazine's page.

In 1988, *DDJ* renewed its commitment to traditional topics—advanced programming in languages such as C, Pascal, Modula-2, Forth, Basic, and operating systems like DOS, UNIX, and OS/2—while expanding its scope to include in-depth coverage of subjects like object-oriented, Macintosh, and real-time programming. And in May, former editor-in-chief, now editor-at-large, Michael Swaine launched a new monthly column titled "Programming Paradigms" which explores techniques that are potential alternatives to conventional programming approaches.

As you turn the pages of this book, you'll see what the future of computing will look like simply by examining the programming resources and issues that are beginning to emerge. Creating photorealistic computer graphics, dealing with multi-language programming environments, and writing portable software for multiple operating system environments are challenges that all programmers will face in the near future.

So, in 1988 *DDJ* did more of what it does the best by providing the kind of in-depth technical information that advanced programmers want and need. This is what you've seen from *DDJ* in the past, and what you'll see more of in the future.

Jonathan Erickson
Editor-in-Chief

Software Availability

All the listings appearing in **Bound Volume 13** are available on two MS/PC-DOS disks. The set costs $19.95. California residents must add the appropriate sales tax.

To order, please send a check made payable to **M&T Books**, or credit card number with expiration date, to:

Bound Volume 13 Listings Disk
M&T Books
501 Galveston Drive
Redwood City, CA 94063

Or, you may order by calling our toll-free number between 8:30 A.M. and 5:30 P.M. Pacific Standard Time: 800/533-4372 (800/356-2002 in California). Ask for **Item #048-6.**

Contents

1	**JANUARY 1988, NUMBER 135**
3	Editorial SARA NOAH RUDDY
4	Running Light MICHAEL SWAINE
5	Letters
7	386 vs. 030: The Crowded Fast Lane TYLER SPERRY
10	A Programmers' Database for the Macintosh Toolbox ABDULLAH AL-DHELAAN and TED G. LEWIS
24	C Chest ALLEN HOLUB
31	To the Macs STAN KRUTE
39	Structured Programming NAMIR CLEMENT SHAMMAS
43	Artificial Intelligence ERNEST R. TELLO
51	Of Interest
53	Swaine's Flames MICHAEL SWAINE
55	**FEBRUARY 1988, NUMBER 136**
57	Editorial TYLER SPERRY
58	Running Light TYLER SPERRY
59	Letters
61	Debugging with the 80386 FRANKLIN GROSSMAN
65	A Serial Protocol Analyzer Program CRAIG A. LINDLEY
91	C Chest ALLEN HOLUB
96	Structured Programming KENT PORTER
99	The Forth Column MARTIN TRACY
103	Examining Room
110	Of Interest
112	Swaine's Flames MICHAEL SWAINE

113 **MARCH 1988, NUMBER 137**

115 Editorial TYLER SPERRY
116 Running Light TYLER SPERRY
117 Letters
119 Object-Oriented Programming and Databases JACOB STEIN
128 A Italic Font in C for the EGA and VGA ANDREW J. CHALK
133 Threaded Binary Trees JAMES MATHEWS
148 To the Macs STAN KRUTE
155 Structured Programming KENT PORTER
158 Examining Room
163 Of Interest
165 Swaine's Flames MICHAEL SWAINE

167 **APRIL 1988, NUMBER 138**

169 Editorial TYLER SPERRY
170 Running Light TYLER SPERRY
171 Letters
173 Creating an Adventurous Language JONATHAN AMSTERDAM
181 Topics in Knowledge-Based Languages BILL and BEV THOMPSON
185 Theorem Proving Using Semantic Resolution ANTHONY J. DOS REIS
200 C Chest ALLEN HOLUB
205 To the Macs STAN KRUTE
210 Structured Programming KENT PORTER
214 The Forth Column MARTIN TRACY
217 Examining Room
220 Of Interest
223 Swaine's Flames MICHAEL SWAINE

225 **MAY 1988, NUMBER 139**

227 Editorial JONATHAN ERICKSON
228 Running Light TYLER SPERRY
229 Letters

231 Developing for the User ROBERT CARR
234 Handling Image Files with TIFF ANTHONY MEADOW, ROCKY OFFNER and MICHAEL BUDIANSKY
242 Virtual Arrays in C MARK TICHNOR
255 C Chest ALLEN HOLUB
259 To the Macs STAN KRUTE
263 Structured Programming KENT PORTER
266 Programming Paradigms MICHAEL SWAINE
273 Examining Room
279 Of Interest
282 Swaine's Flames MICHAEL SWAINE

283 **JUNE 1988, NUMBER 140**

285 Editorial JONATHAN ERICKSON
286 Running Light TYLER SPERRY
287 Letters
289 Writing Real-Time Programs Under UNIX BILL CRAMER
294 Message-Passing Operating Systems DAN HILDEBRAND
301 A Simple Decompiler WILLIAM MAY
311 C Chest ALLEN HOLUB
314 The Forth Column MARTIN TRACY
317 Structured Programming KENT PORTER
322 Programming Paradigms MICHAEL SWAINE
327 Examining Room
332 Of Interest
334 Swaine's Flames MICHAEL SWAINE

335 **JULY 1988, NUMBER 141**

337 Editorial JONATHAN ERICKSON
338 Running Light TYLER SPERRY
339 Letters
341 Getting A Head With HyperAnimation ELON GASPER
349 SQL Development Tools MICHAEL GEARY
355 Pattern Matching: The Gestalt Approach JOHN W. RATCLIFF and

DAVID E. METZENER
359 Getting Down to BASICs BRUCE TONKIN
376 C Chest ALLEN HOLUB
379 Structured Programming KENT PORTER
383 Programming Paradigms MICHAEL SWAINE
388 Of Interest
390 Swaine's Flames MICHAEL SWAINE

391 AUGUST 1988, NUMBER 142

393 Editorial Jonathan Erickson
394 Running Light TYLER SPERRY
395 Letters
397 Speed Trials: Five Cs Compared RICHARD RELPH
404 Find That Function! MARVIN HYMOWECH
408 An Aid to Documenting C STEWART NUTTER
412 A Tool for Secret Key Cryptography JOHN R. MICHENER
416 Objective C ERNEST R. TELLO
432 C Programming AL STEVENS
437 The Forth Column MARTIN TRACY
440 Structured Programming KENT PORTER
443 Programming Paradigms MICHAEL SWAINE
449 Of Interest
452 Swaine's Flames MICHAEL SWAINE

453 SEPTEMBER 1988, NUMBER 143

455 Editorial JONATHAN ERICKSON
456 Running Light RON COPELAND and MICHAEL SWAINE
457 Letters
459 Using Action Charts for Software Development MARTIN STITT
466 Ada for Pascal Programmers K.N. KING
478 Object-Oriented Dimensional Units JOHN A. GROSBERG
485 The State-of-the-Art in Modula-2 KENT PORTER
492 Arguments and Automatic Variables in Assembly Language
RAYMOND MOON

497 XCMD and XFCN: HyperCard's Software Slots STAN KRUTE
510 C Programming AL STEVENS
514 Structured Programming KENT PORTER
519 Programming Paradigms MICHAEL SWAINE
521 Of Interest
522 Swaine's Flames MICHAEL SWAINE

523 OCTOBER 1988, NUMBER 144

525 Editorial JONATHAN ERICKSON
526 Running Light RON COPELAND
527 Letters
529 Adding Extensions to Lisp JONATHAN AMSTERDAM
533 An 80386 Assembler in Forth JOHN B. DILWORTH
537 80386 Protected Mode Initialization NEAL MARGULIS
541 A Double Cross for MASM STEVE HELLER
545 Turbo Prolog 2.0 ERNEST TELLO
557 Structured Programming KENT PORTER
560 C Programming AL STEVENS
564 The Forth Column MARTIN TRACY
566 Programming Paradigms MICHAEL SWAINE
574 Of Interest
577 Swaine's Flames JONATHAN ERICKSON

579 NOVEMBER 1988, NUMBER 145

581 Editorial JONATHAN ERICKSON
582 Running Light RON COPELAND
583 Letters
585 Photorealism in Computer Graphics STEVE UPSTILL
592 Putting Graphical Interfaces into Perspective KENT DAHLGREN
597 Image Compression Via Compilation VICTOR J. DUVANENKO
600 Dynamic Run-Time Structures TODD KING
603 Mapping DOS Memory Allocation ROBERT J. MOORE
608 Inserting Elements into a Basic Integer Array BRUCE TONKIN
610 Prolog/V: Prolog in the Smalltalk Environment

GREGORY L. LAZAREV
630 Structured Programming KENT PORTER
626 C Programming AL STEVENS
633 Programming Paradigms MICHAEL SWAINE
640 Of Interest
642 Swaine's Flames JONATHAN ERICKSON

643 **DECEMBER 1988, NUMBER 146**

645 Editorial JONATHAN ERICKSON
647 Letters
650 Building Software for Portability GREG BLACKHAN
655 UNIX vs. UNIX DONNALYN FREY
659 Writing OS/2 Applications with I/O Privileges RAY DUNCAN
664 Writing Programs for MultiFinder CHRIS DEROSSI
668 Spelunking MS-DOS: Documenting the Undocumented
SCOTT ROBERT LADD
671 EGA and VGA: Smooth Scrolling and Panning ANDREW J. CHALK
674 Find That Function—From Inside Brief! MARVIN HYMOWECH
677 A Class-ier C ERNEST R. TELLO
680 C Programming AL STEVENS
683 Structured Programming KENT PORTER
686 Programming Paradigms MICHAEL SWAINE
703 Of Interest
706 Swaine's Flames MICHAEL SWAINE

709 **Dr. Dobb's Macintosh Special, Summer 1988**

711 Rebirth of the Macintosh RANDY SUTHERLAND
712 A Mac Programmer's Resource Kit ANTHONY MEADOW
720 Overview: Macintosh Programmer's Workshop DAN ALLEN
741 Apple's A/UX: UNIX for the Rest of Us? ANTHONY MEADOW
749 An Introduction to HyperCard Programming DAN ALLEN
762 Editorial MICHAEL SWAINE

765 **Dr. Dobb's Software Engineering Sourcebook, Winter 1988**

767 Editorial PHILLIP ROBINSON
770 Software Engineering Environments STOWE BOYD
778 Software and the Single Programmer T.G. LEWIS
786 Using an API as a Developer Platform JEFFREY M. PARKER
793 Embedded Systems Design—A Special CASE? DAVID KALINSKY and JIM READY
799 Tackling Large-Scale Programming Projects WILLIAM COURINGTON, JONATHAN FEIBER and MASAHIRO HONDA
803 Applying Workstation Technology to CASE DAVID B. LEBLANG
825 Editorial MICHAEL SWAINE

827 **Index**

#135 JANUARY 1988

2.95 (3.95 CANADA)

Dr. Dobb's Journal of

Software Tools

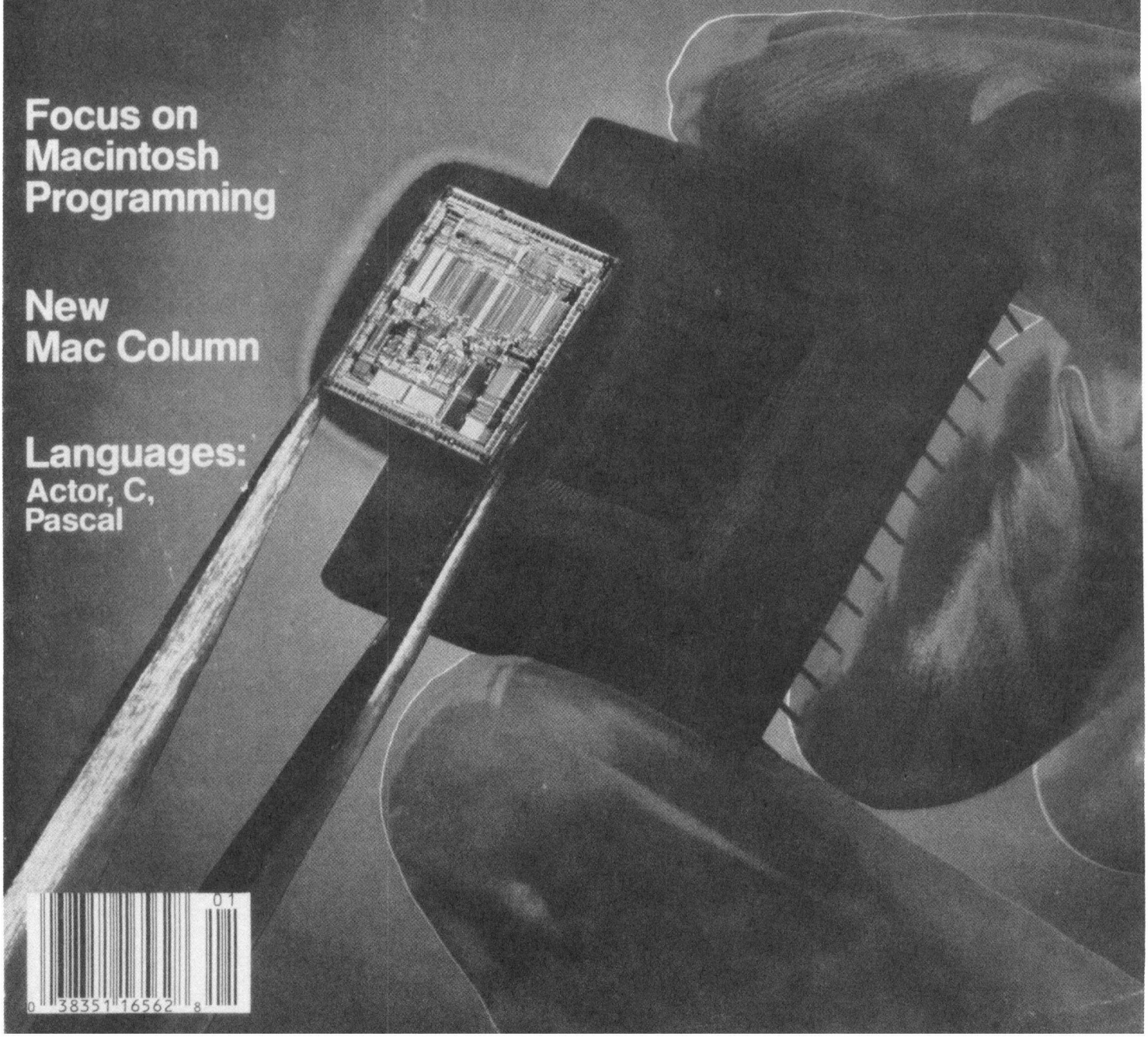
FOR THE PROFESSIONAL PROGRAMMER

The 68030
How Hot Is It?

Focus on Macintosh Programming

New Mac Column

Languages:
Actor, C, Pascal



0 38351 16562 8

ARTICLES

68030 ▶
386 vs. 030: The Crowded Fast Lane **16**
by Tyler Sperry
Skeptical of all the conflicting benchmark data on the Motorola 68030 processor, Tyler set out to discover the truth, and learned that, as usual, the truth is far from simple.

Macintosh ▶
Programmer's Data Base for the Mac **26**
by A. Al-Dhelaan and T.G. Lewis
Macintosh programmers: have you ever wished you could pull down a menu, type in the name of a toolbox routine, and view its Inside Macintosh description on screen? Meet MacMan, the Macintosh manual-in-a-desk-accessory.

Putting ROM Code in its Place **42**
Rick Naro's listing continued from last month.

COLUMNS

C CHEST **72**
by Allen Holub
This month Allen wraps up a two-part presentation of his preemptive multitasking kernel, detailing how the subroutines work.

New Mac column ▶
TO THE MACS **90**
by Stan Krute
In our new monthly column on Macintosh programming, veteran Mac hacker Stan Krute explains how to write custom CDEF resources to create several new species of buttons.

Object-oriented Pascal ▶
STRUCTURED PROGRAMMING **108**
by Namir Clement Shammas
Namir examines the power and problems inherent in an object-oriented programming in Pascal, concentrating on an implementation for the Macintosh. . . . .

Actor ▶
ARTIFICIAL INTELLIGENCE **114**
by Ernest R. Tello
. . . while Ernie looks at object-oriented programming in the PC environment in his critique of version 1.1 of Actor, the object-oriented programming language written by Chuck Duff of Neon fame.

FORUM

EDITORIAL **6**
by Sara Ruddy

Hypertext ▶
RUNNING LIGHT **8**
by Michael Swaine

ARCHIVES **8**

LETTERS **10**
by you

SWAINE'S FLAMES **136**
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX **121**
Ads for ads' sake, or the profit motif

OF INTEREST **130**
Products for programmers

About the Cover
The Motorola 68030. Two googolplex of transistors, count them. Is it the Next Big Thing?

Next Issue
We don't normally comment on unreleased products, but since you are the soul of discretion, we'll tell you this much: the February release (Version 13.2) is now in alpha and will feature tools for debugging. Your SDK should arrive in 30 days and will include reviews of debugging tools and articles on debugging and a new section of short reviews called The Examining Room. The user interface will show minor bug fixes, and the delivery medium will again be high-bandwidth paper. And of course it's all upwardly compatible with the preceding 135 releases.

EDITORIAL

Back in the late 1970s my best friend Renèe gave me a pocket calculator for my birthday. Needless to say, I was impressed. I was sure this modern gadget would make life a lot easier. It was a great little machine: ran on a 9-volt battery, had happy red lighted numbers, and had lots of blue buttons with serious mathematical symbols on them.

Of course, I err in speaking of this machine in the past tense; it still *lives in* my desk at home. I can't bear to throw it away. A few weeks ago my handy little calculator was the butt of several jokes from the very modern *DDJ* editors, and this caused me to pause and reflect a bit as we face a new year full of exciting technological advancements.

Motorola had a big shindig here in October to announce its new 68030 chip and gave out solar-powered calculators as a promotional prize. The guys on the staff generously donated the calculator to me so I could finally enter the 1980s. I appreciate this snazzy new tool, but somehow I can't bear to part with my old clunker.

January 1988. We've reached another year closer to 2000, and I find I have a hard time letting some of what made 1987 possible go without mention. Thanks to those great folks on the *DDJ* editorial staff who have gone on to bigger and better (?) things: Nick Turner, Deborah Hart, Vince Leone, and Levi Thomas. Also our columnists, who we hope will still send in an occasional piece of brilliant prose: Michael Ham, Ray Duncan, and Namir Shammas. (Namir is passing the baton of his Structured Programming column into the able hands of Kent Porter as of next issue.)

We have lots of exciting things in store for 1988, the first being the issue you hold in your hands, our annual 680x0 issue. In this issue we debut our new Macintosh column, To the Macs, by Stan Krute. Our editorial schedule for the rest of the year is as follows:

February—Debugging
March—Object-oriented programming
April—AI languages
May—Designing applications
June—Real-time programming
July—Distributed data (hypermedia)
August—Annual C issue
September—Software engineering
October—PostScript; Forth
November—Graphics and video
December—Operating systems

Give Tyler a call with any article ideas.

Starting in February, we will add Examining Room, a series of short product reviews, to *DDJ*'s traditional fare. We won't accept unsolicited reviews but invite you to join the team of "examiners." If you have any ideas, give Ron Copeland a call.

So now we're ready for a new year. I have my new calculator and you have your new magazine. But, being a sentimental sort, I'm determined to hang on to my relic of a calculator. I picture myself well into the 21st century, cuddled around the heat projection unit with my rosy-faced grandkids, all of them eager to hear another tale of days gone by. "Tell us about your first calculator again, Grandma," they will say. "Well," I'll reply, "Back in the late 1970s your Auntie Renèe gave me a pocket calculator for my birthday. . . ."

As we dive headfirst into a year filled with technological promises, let us not forget the people and forces that brought us to where we are today. Best wishes for 1988.

Sara Noah Ruddy

Sara Noah Ruddy
assistant editor

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Tyler Sperry*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph*
Contributing Editors *Kent Porter, Namir Shammas, Ernest R. Tello*
Copy Editor *Rhoda Simmons*
Production
Director Art/Production *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Fulfillment Coordinator *Francesca Martin*
Book Marketing Mgr. *Jane Sharninghouse*
Subscription Supervisor *Kathleen Shay*
Newsstand Sales Coordinator *Larry Hupman*
Administration
Finance Director *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Advertising
Director *Ferris Ferdon*
Marketing Mgr. *Michael Wiener*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 129*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $27 per year airmail or $10 per year surface. All other countries add $27 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1987 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
V.P. of Publising *William P. Howard*

RUNNING LIGHT

Tyler surfaced from writing this month's lead article just long enough to ask me to step onto this page and tell you what to expect from the arrival of HyperCard. Here goes:

1. Expect to see explosive growth in the number of "light" programmers in the Macintosh environment as a result of HyperCard, well beyond the effect of Turbo Pascal when it burst upon the PC environment. There are several facts that support this claim:

It's bundled. A million people will have HyperCard by the end of this year. Nearly all will actually use it (as an application).

The path from using HyperCard to writing your own programs in HyperTalk is smooth. The stages, from browsing to cut-and-paste application building to modifying existing scripts to writing short scripts of your own to developing full stackware applications with HyperTalk, may be the easiest gradient up to programming ever.

The pressure for a truly easy Mac development tool has been building for four years, and the floodgates are now open.

2. Expect prolific output from these "light" programmers. Again, there are several reasons to believe this:

Ease of programming. Even though it embodies principles unfamiliar to a BASIC programmer, HyperTalk is about as easy as BASIC or Pascal.

The power of the language. HyperTalk objects and messages are generally higher-level components than BASIC statements, so you can do more with fewer of them.

Expected aids to programming. Apple is developing enhancements that will make it easier to do more, including toolkits for controlling serial communications, AppleTalk communications, and interactive video. And there will be more public-domain and shareware programming aids like Andy Hertzfeld's PICT file importer. Commercial products will serve as lessons in programming, since the source is examinable.

3. Deduce from these expectations an overwhelming outpouring of stackware (as it is called). The evidence is already coming in: ten megabytes of shareware and public domain stackware was developed in the first three months after the announcement.

4. Expect the inevitable result: HyperGlut. While some amateurs will produce software that is far from amateurish, we will soon see unprecedented amounts of poor work. But obviously-bad software is just noise in the channel; the real HyperGlut will be in the stackware products that meet needs cheaply but poorly. That will, sadly, be used.

5. Expect the end of the predictable Macintosh user interface. That only worked while nearly everybody conformed nearly all the time.

6. Expect bad practices and poor programming style to become ingrained. This is a more depressing thought the more you dwell on it, because there are more ways to program badly with HyperCard than with BASIC. With HyperCard, snippets of code can reside in various places, such as attached to a button or to the card on which the button resides, and the inheritance structure of HyperCard allows events to drop through various objects, searching for a handler.

The object-oriented equivalent of spaghetti: what a concept.

Michael Swaine
editor-in-chief

ARCHIVES

Reaffirmation

"We who have had some degree of involvement with *DDJ* and People's Computer Company (PCC) modestly think of this publication as the lever which, with the slightest degree of pressure, just might move the world a bit. By serving the high end of the technological spectrum, some of our efforts do find their way by mysterious means into products and services which help people. This happens when one of our readers who sees the true potential of computers, and some piece of software we publish, puts them together in new ways." *Editorial, Marlin Ouverson, Editor,* DDJ, *December 1982.*

Tales of Future Passed

"The Xanadu Hypertext System is one of the most powerful systems in existence for managing text in a micro-to-mainframe database environment. It can store arbitrary numbers of documents and retrieve them on demand. It *can organize* them by using a highly generalized link facility and by allowing (and keeping track of) multiple users on the same system, and allow those users to work with each other on a common document base. It can grow indefinitely over a large distributed network with minimal degradation in performance. . ." Roger Gregory, *"Xanadu - Hypertext from the Future,"* DDJ, *January 1983.*

Pretty Is As Pretty Does

"The laws of entropy insure that the line numbers of a debugged and operational BASIC program give the appearance of having been selected by a KENO machine. In fact, while several texts detail how the boundary conditions of a KENO game lead to predictable outcomes, finished programs seldom exhibit this property. Many a time I have spent an extra hour retyping a finished program while spacing the line numbers evenly just to make it look good." *"Renumbering & Appending BASIC Programs on the Apple II Computer," Steve Wozniak,* DDJ, *February 1978.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Optimum Performance?

Dear *DDJ*,

Either Richard Relph's article on optimizing compilers for C (August 1987) has a slight error, or I'm going to have a lot of trouble with my code if I start to use one. I believe the example of common subexpression elimination he refers to as block range optimization is at least module range and could be program range.

Under the K & R definition of C, a two-dimensional array is not necessarily a continuous block of storage. An expression such as *a[i][j]* refers to the *j*th element of the block pointed to by *a[i]*. The element *a* is an array of pointers to appropriate objects. This permits arrays in which rows can have different sizes and in which rows can be switched by swapping pointers and permits arrays to be built up dynamically during execution. Consequently, the identification of *d[i][j]* with *d[0][t0]* where $t0 = i*10 + j$ is not always correct.

Even if we can be assured that the dimensions of *d* will be 10×10 when *copy( )* is called, if the storage of *d* was allocated one row at a time (as is often the case in my programs), there is no assurance that *d[i][j]* can be addressed as in the example. There is even no assurance that the address of *d[i][j]* will bear the same relationship to that of *d[0][0]* as *s[i][j]* will to *s[0][0]*. In the particular example given, that assurance is provided by the declarations of *d* and *s*. (Even that assurance could be compiler implementation dependent, though I know of no compilers for which it would not be correct.) But these were globals, declared outside the function, so the optimization indicated requires at least module-level information and in other cases might require program-level information.

Any optimizing compiler that streamlines multidimensional array addressing in this way will either have to incorporate information beyond the function being optimized or it will have to restrict this technique to certain arrays declared within the function.

Thanks for an interesting article. Keep up the good work.

Clyde Schechter
116 Pinehurst Ave., Apt. D-63
New York, NY 10033

Better (or Worse) Standard

Dear *DDJ*,

I had mixed emotions after reading Richard Relph's description of the forthcoming ANSI C standard in the August 1987 issue. As a programmer working almost exclusively in C, I like the idea of replacing local dialects with a common language, and I like many of the proposed enhancements. I am disturbed, however, by the move to merge the library routines into the definition of the language. At present, the C language is unique in not including library routines as part of the specification, which I think is the right approach.

For example, the article mentions that library function names and macros are to be reserved words, to eliminate the possibility of programmers substituting their own functions for standard ones. I have often substituted such routines to good effect while debugging a program, however. Similarly, it can be useful to set breakpoints on routines such as *strcpy( )*, which will be impossible if the compiler is free to substitute in-line code for my subroutine call.

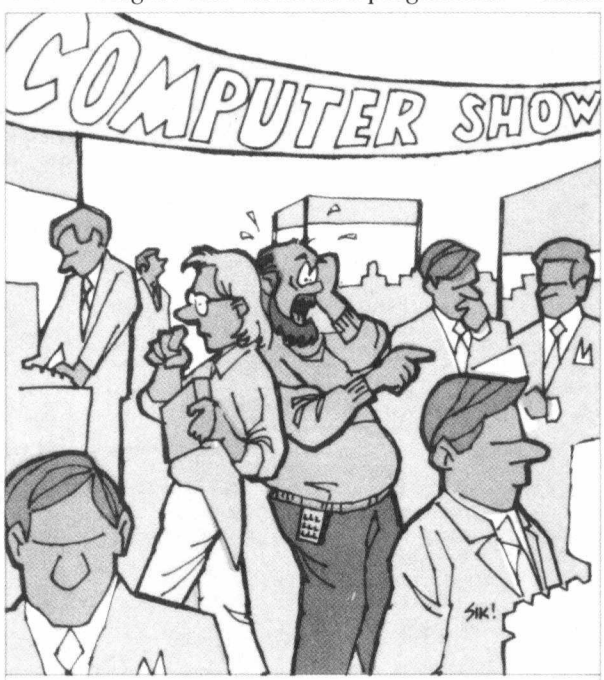
Although I agree that the library functions should be standardized, making them part of the syntax itself hurts the language. The justification for this seems to be that it makes it easier to optimize the code, but it is a poor trade to take away capabilities in order to go easy on the compiler!

At present, I know what's going on when I call *strcpy( )* in any compiler, and if I want to speed up my program, I can write in-line code myself. With ANSI C, programmers won't know how to write efficient code without knowing which compiler will be used to compile it. This improves portability? Perhaps the ANSI committee needed a few more members who buy compilers along with those who write them.

William F. Linke
286 Dunhams Corner Rd.
East Brunswick, NJ 08816

Richard Relph replies:

Mr. Linke's letter makes a key observation. The C library is now "standardized." Without this change, program portabil-



*"Good Grief—now **They've** penetrated **Our** networks!"*

ity is unobtainable. It should be noted, however, that library routines are not mandated for freestanding environments, such as embedded systems.

I have some difficulty with his final paragraph. He seems to be wanting both maximal speed and maximal portability, a desirable goal, I agree. The two are often at odds with one another, however. Perhaps a solid example would help here. Suppose I need to perform a string copy operation. I could either call *strcpy* or code it in-line. Mr. Linke's assumption is that, by coding it in-line, he would get the best performance. This is not necessarily true, even with today's non-ANSI compilers. Most libraries implement *strcpy* in assembly language, using instructions that most C compilers cannot normally generate, such as *REP MOVSB* in the 8086. Therefore, the programmer already doesn't know what's most efficient. Mr. Linke further assumes that he "knows what's going on when I call *strcpy( )* in any compiler." Is he aware that some existing compilers perform in-line expansion of key library functions? MetaWare and Microsoft's both do this, and I'm sure others will follow.

By reserving the library function names for the compiler, the compiler can do several things that would otherwise be impossible. First, the library itself can call library routines. If users were allowed to replace *strcpy*, for example, other library routines such as *printf* could not call it. This essentially means that for the normal case when programmers do not replace *strcpy*, but do call it, the library must include two copies, one with some strange name for the library itself and one for the user to call.

Second, the library can be granularized at other than the function level. Although it is nice to have a library that is function granular, some machines cannot support libraries with so many "chunks." If the implementor thought that *strcpy* and *memcpy* made a reasonable pair and put them in a single module, and the programmer replaced *strcpy* but called *memcpy*, what could the linker do?

Last, the compiler can now replace some calls to functions with in-line assembly-language code, taking into account the actual parameters. This is optimal efficiency, allowing the insertion of the assembly-language version of *strcpy* in the code, saving the call and return overhead.

At the risk of being taken literally and in the extreme, let me say that leaving operations at as high a level as possible is always desirable. If I can call *strcpy*, or write it in-line, I will pick the call—for efficiency, for maintenance, for clarity, and for standardization.

Finally, let me say that these compiler features are what people will probably start basing their buying decisions on. This is a net plus. People who want what Mr. Linke wants will probably be able to get it and people who don't will be able to get something else.

```
254 10         UNITS INCHES
254 12 *  10   UNITS FEET
254 36 *  10   UNITS YARDS
10  1          UNITS CENTIMETERS
1000  1        UNITS METERS

\ Usage:
10 FEET   . <cr>  3048  ok
3 METERS  . <cr>  3000  ok
\ ......................
\ etc.
```

Example 1: *Conversion program using the defining word* UNITS

```
VARIABLE  <AS>     0 <AS> !
: AS  -1 <AS> ! ;
: UNITS  CREATE  SWAP , ,  DOES>  D@  <AS> @
         IF  SWAP  THEN  */  0 <AS> ! ;
BEHEAD' <AS>     \ TO MAKE IT LOCAL FOR SECURITY

\ UNIT DEFINITIONS REMAIN THE SAME.
\ Usage:
10 FEET . <cr>  3048  ok
3048 AS FEET .  <cr>  10 ok
```

Example 2: *Code to convert back to input units when outputting*

Dimensional Data Types in Forth

Dear *DDJ*,

This letter is an addendum to Eric Lundquist's comment (Letters, August 1987) on Do-While Jones' article "Dimensional Data Types" (in Ada). His astonishment at the lengthy code required to implement this simple idea in Ada is, of course, justified. As a theoretical physicist and accident consultant who uses dimensioned data every day of my life, I was equally astounded.

Experienced Forth programmers, on the other hand, must have smiled indulgently, murmured "What fools these mortals be!", and passed on. As a fairly recent convert to Forth (and a heavy user of this language for scientific computing), however, I cannot pass up this opportunity (we recent converts tend to be zealots) to exhibit how easily dimensioned data can be implemented in Forth. Perhaps Mr. Lundquist—who feelingly alludes to the advantages of assembly language—will then be sufficiently piqued to give Forth a try.

Suppose you need to input distances in various units. For example, U.S., English, and Continental police reporting accidents *might wish to* use inches, feet, and yards, or centimeters and meters. Rather than writing different versions of a program, and making users worry about which one they are using, it is simpler to write one program and make unit conversions part of the grammar. Thus, for example, you might keep all internal lengths in millimeters and convert as follows:

```
: INCHES 254 10 */ ;
: FEET [ 254 12 * ] LITERAL 10 */ ;
: YARDS [ 254 36 * ] LITERAL 10 */ ;
: CENTIMETERS 10 * ;
: METERS 1000 * ;
```

The usage would be:

```
10 FEET . <cr> 3048 ok
```

These are more definitions than necessary, of course. A better alternative

(continued on page 126)

386 vs. 030: The Crowded Fast Lane

Picking the fastest CPU can be difficult when not even the benchmarks agree

by Tyler Sperry

Past issues of *DDJ* have detailed the introduction and rise of Intel's 80386 microprocessor and its related software in some detail. The ability of 386 machines to maintain compatibility with MS-DOS software while also giving substantially faster performance is something we all can respect. The 80386 isn't the only CPU that can lay claim to the title of fastest microprocessor, however. Recently, Motorola introduced the latest member of its 68000 line—the 68030—with performance claims that would leave the 80386 in the dust. Obviously, this claim bears some investigation.

The 68000 Gets Respectable

Like many of you, my first experience with a 68000-based computer was the original 128K Macintosh. *Disillusionment* is the mildest term I'd use for my first encounter. Despite the promise of a superfast processor with lots of 32-bit general-purpose registers and plenty of memory (128K!), I was able to go back to my CP/M machine without regret. By burdening the CPU with updating the video and emulating a disk controller, the Mac seemed a perfect demonstration of Grosch's Law[1]; the CPU might be inherently faster than 8-bit machines, but you'd never be able to prove it by the performance.

In the last few years we've seen the introduction of a number of machines that have delivered on the promise of the 68000. The Mac line has matured to produce the 68020-equipped Macintosh II that rivals the performance of the IBM PC AT. At the other end of the spectrum, Sun Microsystems has had enormous success with Unix boxes based on the 68000 and its descendants. Indeed, once you subtract a few proprietary CPUs (from IBM and DEC), workstations are powered almost exclusively by the 68000.

When not programming or reviewing hardware and software, Tyler Sperry works at his day job as Editor of DDJ.

The Paradox of the Installed Software

Fine, you say, but what difference does that make when there are more than 8 million DOS machines out there? And what about the 80386? Doesn't it blow away the 68020?

On the first point, I am happy to say that this article is concerned with comparing the latest offerings of the Intel and Motorola lines, not with software and hardware marketing. Still, those questions often come up in a discussion of the technical merits of competing CPUs. (The AMD 29000 looks to be a fantastic chip, but when will you get to program one?) For now, let it suffice to say that:

- Any machine hoping to become the new standard in the personal computer marketplace will have to address the huge software "inertia" of the millions of DOS machines and their software compatibility demands.
- The high-performance CPUs coming out in the next few years will probably make this point moot by providing PC emulation at speeds meeting or exceeding an AT. (This software already exists for Unix boxes.)
- Even if software emulation isn't suitable, 286 "clone cards" are becoming increasingly easier to come by on a variety of buses.

Apples and Oranges

At first glance, it might seem simple to compare the performance of the 80386 and the 68030. Just set up a test jig with some memory and the two CPUs and run some benchmarks. Although that approach might have worked with 8-bit CPUs (the infamous "good old days"), there are a few problems when you get to the new 32-bit processors.

There's memory, for example. Do you furnish the processors with the fastest static RAM available and let them run flat out, or do you run with a "typical" system

(continued from page 17)

(dynamic RAM) and slow the CPUs down with wait states? Arguments can be made on either side.

In the case of the 68030/80386 controversy, this concern has actually been addressed by the chip manufacturers. Both CPUs have provisions for handling the common problems associated with using dynamic RAM slower than the CPU can handle. Both chips have a provision for "burst mode" access, for example, which allows contiguous bytes of memory to be accessed without the delays normally associated with address setup and decoding. In some respects these two CPUs are more similar than different. There is one significant difference in their approach to handling memory access, however, that has a substantial impact on performance.

Cache As Cache Can

Let's take a brief detour down memory lane (so to speak). Back in the days of the 6502 and 8080, access to memory was slow but relatively straightforward. If the processor wanted an instruction, it went out to the memory bus and fetched one.

This procedure began to change with the introduction of the Intel 8088. One of the features of the 8088 was a 4-byte prefetch instruction queue that attempted to separate memory bus activity from computation time. Program instructions were moved from memory into a prefetch queue and then acted upon. Although this sped things up a bit, it was of limited usefulness. (See this month's Letters for more on the subject.) Indeed, the less charitable have referred to the prefetch queue as the prefetch bottleneck.

Eager to please, the engineers at Intel improved things in subsequent Intel designs: the 80386 has a 16-byte prefetch instruction queue. (A simplified schematic is shown in Figure 1a.)

Motorola's attempts at speeding things up became noteworthy with the introduction of the 68020. The 68020 does not have an instruction queue but rather a 256-byte instruction cache. Once an instruction is loaded into the cache from memory, it need not be reloaded unless it's been replaced by a more frequently used instruction. Thus, a small, tight loop can run entirely from on-board cache memory and result in much faster performance. An instruction queue, on the other hand, is by definition limited to operating as an instruction pipeline; any branch taken forces the reloading of the queue.

As you might expect, the addition of an instruction queue can substantially improve a processor's performance. The amount of improvement will, of course, depend on how many tight loops there in your code. (Yet another reason to be wary of small benchmarks.) Thayne Cooper and some engineers at Sperry ran both the 68020 and 80386 through some modified *EDN* benchmarks and published the results in *IEEE Micro*.[2] While a 16-MHz 80386 was able to surpass a 16-MHz 68020 with a disabled cache, enabling the cache better than halved the original 68020 benchmark times. (The cached 68020 beat the 80386 in all tests except the string search benchmark.)

Lies, Damn Lies, and Benchmarks

Now, given those benchmarks results, it'd seem pretty clear cut. The performance improvement of boosting the clock speed to 20 MHz should be pretty much the same for either chip. Score them neck and neck—with the edge to the 68020—and we're done, right?

Alas, as my friend Jerry Pournelle would say, it isn't all that simple. The benchmarks done by Cooper were modified 16-bit *EDN* benchmarks, performed on special hardware. The hardware was designed to keep things as

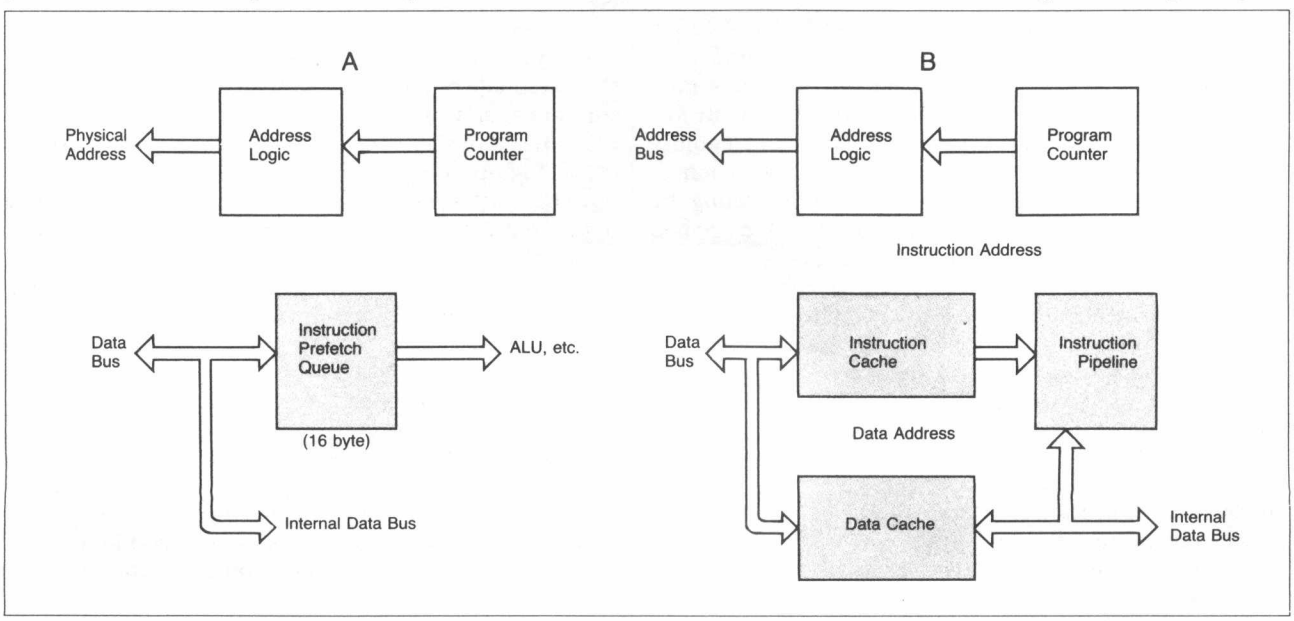


Figure 1: *Simplified view of the memory interface for the 80386 and 68030. The 80386 (a) has a 16-byte prefetch instruction queue. The 68030 (b) features a modified Harvard architecture with separate 256-byte caches for both instructions and data.*

(continued from page 18)

equal as possible for the various processors (the 32032, 32100, and the 80286 were also tested).

Unfortunately, life isn't always fair: your choice of machines will often not include units comparable in all aspects except CPU; sometimes the benchmarks used in a test don't always bear a strong resemblance to your actual application and environment; and the compiler used can impact your results tremendously.

Consider the case of poor Richard Grehan at *Byte*.[3] He took several varieties of 386 computers and accelerator cards, compiled some benchmarks, and ran them. Then he did the same thing for a Mac II and some 68020 accelerator cards in different environments. If you have some experience with benchmarks (or if you read *Byte* regularly), you can anticipate what he found: the 80386 outperformed the 68020 in the majority of tests.

How to explain this? Well, there are some things to note in the *Byte* article benchmarks. First of all, these tests were performed with the intent to test mathematical performance. The only nonmathematical tests were the infamous Sieve and a quicksort routine. As the commercials say, your actual milage may vary.

Second, although Grehan tried to use the same compiler vendor for all machines, this wasn't always feasible. Some of the compilers used for two of the 68020 machines gave substantially better times than the Macintosh compiler used for the other tests, and in fact these times were in the same neighborhood as the best 80386 time (a 16-MHz Compaq 386 with an 80387 coprocessor, in case you were wondering).

The lesson here is unfortunately all too clear. The best benchmark is your target application, ported to the prospective machine. Depending on the optimizations offered by the compiler and individual machine peculiarities, you'll find benchmarks vary widely—there are too many confounding variables for a categorical statement that one chip is better than another. Still. . . .

Going Back to School

After all that discussion and equivocation on the subject of the 68020 vs. the 80386, you'd expect making a clear statement on the relative performance of the 68030 wouldn't be too plausible. After all, as of this writing, there aren't many 68030 machines available to test. (Both Apple and NeXT are rumored to be working on 68030 designs; both refuse to comment on unannounced products.) In reading through the literature, though, I came across some things that can let us make a pretty good guess.

To start with, the 68030 implements a modified Harvard architecture along with expanded caching. A Harvard architecture machine uses separate address and data buses for both instructions and data; in the 68030, a modified Harvard scheme is employed, in which separate buses are used internally and then multiplexed for access to the system. Figure 1b, page 18, shows a simplified schematic of the 68030's memory interface.

Notice that there are now two 256-byte caches: one for instructions and one for data. Given the radical improvement a cache made in the 68020's performance, you can see why Motorola is proudly trumpeting the 68030 as "twice the microprocessor." Of course, it didn't hurt that the chip runs at a clock speed of 25 MHz.

Conclusions

Given that the 80486 is still quite a ways away, Intel would probably like you to believe that a 16-MHz 80386 is equal to a 20-MHz 68020 and that its 20-MHz 80386 is equal (or better) to a 25-MHz 68030. Motorola, as you might expect, has a different view: a fast 68020 is a match for any 80386 and the 68030 blows away an 80386 at any speed.

Aside from the engineer's instinctive distrust of (other people's) benchmarks, and despite the vendor charges and countercharges concerning the benchmarks, there are some clear lessons:

- Other things being equal, an 80386 and a 68020 will perform at roughly the same rate: bloody fast.
- A 68030 at 25 MHz will probably be faster than any 80386 you find. How much faster, though, will depend a great deal on your software and compiler.
- If your application is primarily number crunching, a fast math coprocessor is essential and its presence or absence will probably swamp other aspects.
- A weak compiler can mislead you on the performance of a given system. Conversely, a highly optimizing compiler can completely destroy the value of a poorly constructed benchmark.[4]
- Beware of virtual machines. Today's 5-MHz PC clone is faster than a 50-MHz 80486 box that won't be shipping for another six months.

Notes

1. Ted Nelson, *Computer Lib/Dream Machines* (Microsoft Press, 1987).
2. T.C. Cooper, W.D. Bell, et al., "A Benchmark Comparison of 32-Bit Microprocessors" *IEEE Micro*, vol. 6, no. 4 (August 1986).
3. Richard Grehan, "The New Generation: A Closer Look" *Byte* (September 1987).
4. Richard Relph, "Optimizing Compilers for C" *DDJ*, vol. 12, no. 8 (August 1987).

Bibliography

Motorola. *MC68030 User's Manual*. Motorola, 1987.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

A Programmers' Database for the Macintosh Toolbox

by Abdullah Al-Dhelaan and
Ted G. Lewis

One challenge to writing application software for Apple's Macintosh is the complex environment. Programming the Mac requires the use of an extensive set of ROM routines known as the Toolbox. These routines, although largely responsible for the machine's radical ease of use, add appreciably to the Mac programmer's work load. The Toolbox contains more than 600 routines, and many of them are required in writing even the smallest application. Most programmers find themselves continually referencing Apple's massive documentation, *Inside Macintosh*, for technical details on the numerous procedures, functions, and data structures of the Toolbox.

At first, using *Inside Macintosh* as a handy reference manual might seem a minor chore. It has a good index and is divided into five logical volumes. Unfortunately, information retrieval time becomes a dominant factor in the development process, and it quickly becomes obvious that an on-line, electronic means of retrieval is needed.

Our program, MacMan, is an on-line programmers' database that contains much of the key information found in *Inside Macintosh*. Programmers can use MacMan to fetch the name, parameters, comments, and often-needed descriptions of the Toolbox routines. This information can be accessed from within an editor as an aid to documentation or can be used simply as a way to acquire a better understanding of the inner workings of the Mac. Currently, the database contains more than 500K of text describing the Toolbox routines as well as many other useful facts about the Macintosh.

Putting* Inside *Macintosh* inside *your Macs

The design goals for MacMan were simple: the program should be convenient to use, and it should contain useful information. Above all else, we knew that it had to be convenient to be compelling. Convenient meant both simple and fast. We knew that programmers wouldn't accept MacMan if they had to learn a new language or come to grips with a complex database system.

We also knew that the information contained had to be accurate and useful. We could have written the information in a more useful form than that of *Inside Macintosh*, and we could have selected information from various other sources. We rejected these alternatives, however, because we didn't want to risk introducing errors or to accidentally include ambiguous passages.

We therefore elected to copy carefully selected passages directly from *Inside Macintosh*. We worked hard to convince Apple Computer to let us use the copyrighted contents of *Inside Macintosh* specifically to avoid errors or misrepresentations of fact.

MacMan is not a generalized programmers' database program. We willingly sacrificed generality for user convenience. As a result, MacMan is both fast and extremely easy to use. The description of any Toolbox routine can be retrieved from the 500K database file and displayed in a window—all in less than a second.

Using MacMan

There are a two ways to use the database: an abbreviated version of MacMan can be installed in the system file as a desk accessory (DA), or a full-fledged application version of the program can be launched from the desktop.

A DA is a special program that can run concurrently in memory with another application. Most *DDJ* readers are familiar with the concept of DAs from TSR (terminate-and-stay-resident) programs such as SideKick, but these products illustrate an ad hoc solution to the problem of writing a DA. In contrast to the PC, the Mac allows DAs to be integrated into the Macintosh operating environment. It's this integration that we'll be emphasizing in this article. We'll describe the MacMan desk accessory and then show how it was implemented using the Macintosh Toolbox functions.

Two simple access methods are provided through the menu: Find by Name and View by Category (see Figure 1, page 25). Find by Name, as you might expect, retrieves the desired Toolbox routine by its name. If you don't know the routine's name, you can select the View by Category menu item (see Figure 2, page 25). View by Category lets you select one of the 28 managers as a category

Abdullah Al-Dhelaan and Ted Lewis work in the Computer Science Dept. Oregon State University, Corvallis, OR. 97331.

and then browse through it. When you browse through a category, the Toolbox routine names are displayed in alphabetical order—selecting one of them results in a full display of the routine's description (see Figure 3, below). If MacMan is unable to find a Toolbox routine, it reports the error and asks if it should browse all the routines that begin with the same letter (see Figure 4, page 26).

The subject of this article, DAMacMan, is a version of the database that runs as a desk accessory. Before delving into the inner structure of DAMacMan, however, we'll first review the structure of the typical Macintosh application and how it relates to calling DAs.

The Structure of Mac Applications

Every Macintosh application consists of at least one event loop that determines what operations the application's user is allowed to perform. The event loop must handle all user interactions such as mouse clicks, menu selections, and icon manipulations. The existence of an event loop makes Macintosh applications resemble real-time control programs more than traditional interactive sequential programs.

Every well-behaved application must include a Toolbox call to *SystemTask* so that periodic actions, such as updating the system clock, can be performed by the Macintosh operating system. In Example 1, page 26, we show only one *SystemTask* call for each pass through the main event loop, but in general *SystemTask* should be called at least once every 16 clock ticks (a tick is defined as a 60th of a second). If the application is doing a lot of work on each pass of the event loop, then the *SystemTask* call should be made more often.

The application calls *GetNextEvent* each time through the event loop in order to find out what events have taken place since the previous pass. *GetNextEvent* calls the *SystemEvent* routine, which simply intercepts the stream of events, and if an event belongs to the DA, that event is shuttled to the DA rather than to the application.

Suppose the user selects the DA menu and presses the mouse button; this will cause a *mouse-down* event (*mouseDown*) to be generated by calling the *DoMouseDown* routine. The application programmer must write the *DoMouseDown*

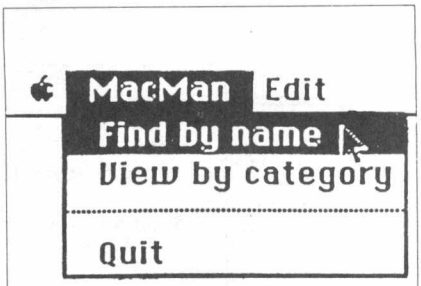


Figure 1: *A Toolbox routine is retrieved by giving either its name or its manager category.*

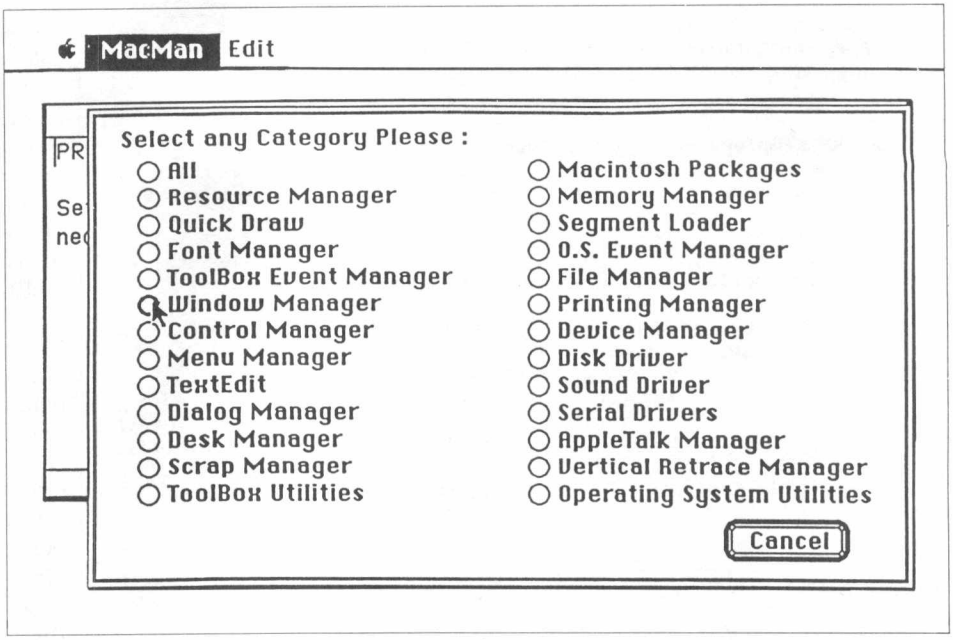


Figure 2: *The MacMan catagories cover the ROM routines for both the user interface Toolbox and operating system.*

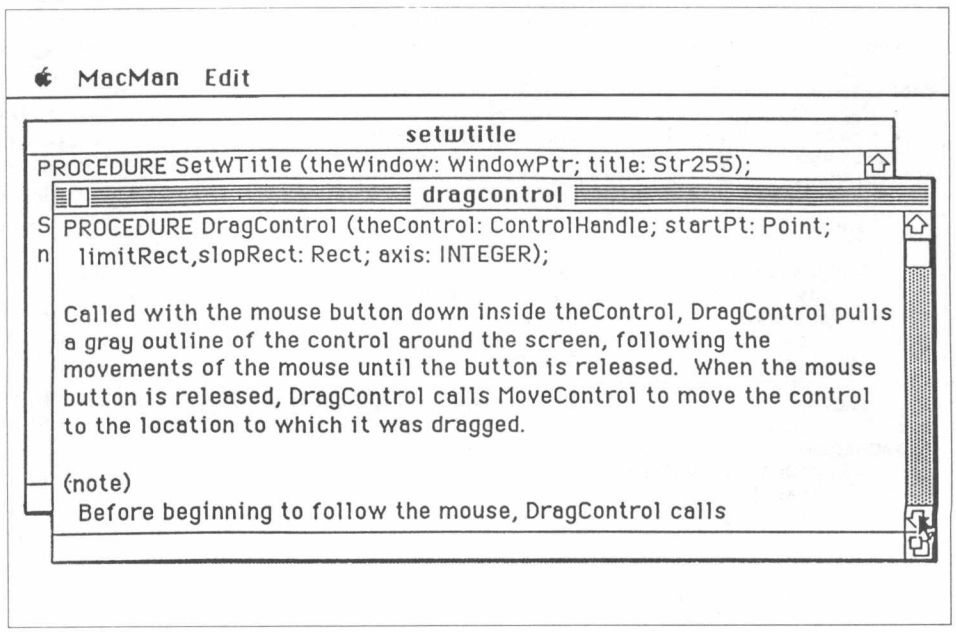


Figure 3: *An example of a displayed Toolbox routine*

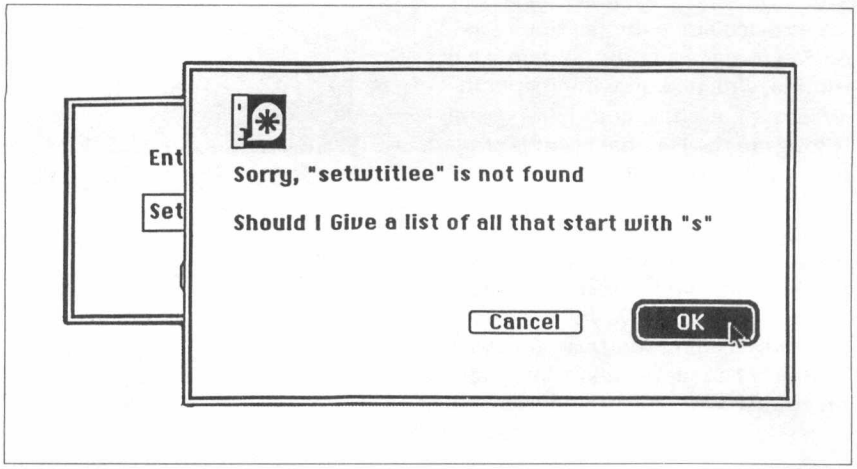


Figure 4: *MacMan reverts to browsing if the named Toolbox routine can't be found.*

```
PROCEDURE SimpleEventLoop( Var Event: EventRecord );
   Var UserAction     : Boolean;      {Has the user done something??}
       Finished       : Boolean;      {Exit When User Quits}
   Begin
       Repeat
       SystemTask;                    {To support periodic events}
       UserAction := GetNextEvent( AllEvents, Event);
                                      {To invoke SystemEvent}
       If UserAction then             {Handle the event...}
       Case Event.what Of
                 mouseDown: DoMouseDown(Event, Finished );
                 KeyDown    : DoKeyDown(Event, Finished );
                 ActivateEvt: DoActivate (Event, Finished );
                 UpDateEvt  : DoUpdate(Event, Finished );
       End;      {Case}
       Until Finished;                {terminate the program?}
End;                                  {SimpleEventLoop}
```

Example 1: *A simple event loop*

```
PROCEDURE DoMouseDown(      Var Event      : EventRecord;
                            Var Finished: Boolean );
Var
   whichWindow : WindowPtr;{Window that mouse was pressed in}
   whereIs     : INTEGER;      {Part of screen where mouse
                                                was pressed}
Begin {DoMouseDown}
   {Where on the screen was mouse pressed?}
   whereIs := FindWindow ( Event.where, whichWindow);
   Case whereIs Of
      InDesk:                 {In Empty Space...}
         {Do nothing};
      InMenuBar:              {Menu Selection...}
         DoMenuClick;
      InSysWindow:    {Aha! In a DA... }
         SystemClick ( Event, whichWindow);
      InContent:              {In Application's window...}
         DoContent (whichWindow);
      InDrag:                 {Drag Application's window...}
         DoDrag (whichWindow);
      InGrow:                 {Resize Application's window...}
         DoGrow (whichWindow);
   InGoAway:          {Close Application's window...}
         DoGoAway (whichWindow)
   End  {case}
End; {DoMouseDown}
```

Example 2: *The DoMouseDown routine*

(continued from page 25)

routine in such a way as to call the appropriate DA. The code necessary to do this is shown in Example 2, below.

The events that are diverted from the application to the DA are channeled into the DA processing code in two ways, as shown in the *DoMouseDown* procedure. The first way is through the *SystemClick* Toolbox routine, as shown in case *InSysWindow* of *DoMouseDown*, and the second way is through a menu selection.

When a mouse-down event occurs in a system window, the application code should call *SystemClick*. If the mouse-down event is in a DA window, *SystemClick* takes care of processing the event instead of the application. This case will be discussed later as it is what happens when the DA is already on the screen.

The mouse-down event could also occur in the DA menu (under the apple), which would mean that the DA is to be activated (this is called opening the DA). This is the case we are most interested in for the time being. If the user has selected the MacMan DA, for instance, then the application program must handle the opening of the DA from the *DoMenuClick* routine, shown in Example 3, page 30.

As shown in *DoMenuClick*, when an application calls the *MenuSelect* (or *MenuKey*) Toolbox routine, a call is made to *SystemMenu*, which passes the event to the DA (if the event is a mouse-down in the DA menu). The DA must then handle the event and return a zero to the application. Otherwise, *MenuSelect* returns a long integer containing the menu number in its *HiWord* and the item number in its *LoWord*. In this case, when the user selects the Apple menu, and within this menu, the MacMan DA, the *DoAppleChoice* routine (see Example 4, page 30) is called to activate the DA.

This sequence of actions opens the DA and prepares it for use alongside the currently running application. In summary, the sequence is:

- A mouse-down event occurs and

(continued from page 26)

is handled by the event loop.

- The *DoMouseDown* routine decides the event is *InMenuBar*.
- The *DoMenuClick* routine decides the event is an *AppleMenu* event.
- The *DoAppleChoice* routine decides the event is a DA selection.
- The name of the DA (*accName*) is obtained and the DA opened.

The Structure of a Mac DA

A DA is a "mini-application" that can be run concurrently with a Macintosh application. A DA cannot exceed 32K of executable machine code and data and is installed in the start-up disk system file using a Macintosh utility program called the Font/DA Mover. Once a DA is installed, it no longer has a file or an icon visible from the desktop. Instead, the user opens a DA by selecting it from the standard Apple menu, which by convention is the first in the menu bar.

After a DA has been opened, its window, if any, is displayed on the desktop and it becomes the active window. To close a DA, the user can click the DA's close box (in its own title bar), or another program can call the function *CloseDeskAcc* to close it. The DA will then disappear and the frontmost window will become the active window.

A desk accessory may have a menu of its own, which will be added to the menu bar when it is active and deleted when it is not. The *Cut*, *Copy*, and *Paste* commands in a standard Edit menu can be used by an active DA. They are very useful for copying and pasting between the DA and the application or another DA.

Writing a DA

Writing a DA is a lot more difficult than writing a "plain vanilla" application because a DA has no main procedure, no main event loop to obtain events, and no global variables.

Technically, a DA is known as a Macintosh device driver, and each DA is required to have three special procedures: *Open*, *Close*, and *Ctl* (for control). These procedures are called directly by the operating system through a special table called the DA Header.

Each of these procedures requires two formal parameters of type *Device Control Record* and *Parameter Block Record*. A *Device Control Record* is created when a DA is opened and destroyed when it is closed. A *Parameter Block Record* is created by the operating system each time any of the three routines is called. It is used to inform the DA about the purpose of the call.

The Macintosh operating system calls *Open* whenever a DA is selected from the Apple menu and calls *Close* whenever the close box on the title bar of the DA's window is clicked or *CloseDeskAcc* is called by some other program. Between these two calls, the system will call *Ctl*.

What DA Procedures Do

When the *Open* procedure is called to open the DA, it will:

1. Create the DA window, if there is one
2. Set the *WindowKind* field of the window's *WindowRecord* to the DA's driver reference number. (This field is set so the operating system can call the correct DA when an event

MAC DATA BASE

occurs in a DA window.)
3. Set the *DctlWindow* field of the DA's *Device Control Entry Record* to the window pointer.
4. Allocate space for global data in the field *dctlstorage* of the *Device Control Entry Record*.

```
PROCEDURE DoMenuClick;        {  Handle mouse-down event in
                                                    menu bar.  }
Var
   menuChoice         : LONGINT;      {Menu ID and item number}
   theMenu    : INTEGER;      {Menu ID of selected menu}
   theItem    : INTEGER;      {Item number of selected item}

Begin {DoMenuClick}

   menuChoice := MenuSelect (TheEvent.where);
      if menuChoice <> 0             {Application, or DA?}
      Then begin                     {Application...}
          theMenu := HiWord(menuChoice);{Get menu ID}
          theItem := LoWord(menuChoice);{Get item number}
          Case theMenu Of
              AppleID:       {Make selection from Apple menu}
                  DoAppleChoice (  GetMenu( AppleID ), theItem);
              FileID:        {Make selection from File menu}
                  DoFileChoice  ( GetMenu( FileID ), theItem);
          EditID:     {Make selection from Edit menu}
                  DoEditChoice  ( GetMenu( EditID ), theItem)
   End;   {case}
   End;   {if}
End;          {DoMenuClick}
```

Example 3: *The DoMenuClick routine*

5. Initialize the global data.

Close is called to close the DA. It first disposes of its window and stores nil in the *DctlWindow* field of the *Device Control Entry Record*, then it disposes of any global data it might have allocated in the *dctlstorage* field of the *Device Control Entry Record*.

Ctl is called to enable the DA to handle the action indicated by the *csCode* field of the *Parameter Block Record*. There are nine such actions, as shown in Table 1.

Passing Text

An application must have the standard Edit menu if it wants to support passing text to and from DAs. The order of items in this menu is important, but the menu can be made longer by adding items at the end.

A DA program consists of at least three special procedures called Open, Close, and Ctl.

The standard Edit menu contains Undo, Cut, Copy, Paste, and Clear. When a user chooses one of these commands, the application must call Toolbox routine *SystemEdit* from within *DoEditChoice* (see Example 5, page 30). The menu items are numbered 0 through 5 internally, which is why we subtract 1 from the item number (*theItem–1*) in the routine shown in Example 5. If the active window is a system window (that is, a DA window), then *SystemEdit* will return false and the application will process the command as usual. Otherwise, *SystemEdit* will shuttle the event on to the DA for command processing and return true.

Resources for DAs

The code for the DA is not a *CODE* resource, as it is for applications, but is a *DRVR* because a DA is actually a device driver. The Macintosh resource compiler RMaker can be used to create a *DRVR* resource by reading the *CODE* resource created

1. AccEvent	An event (update, activate, keyboard, . . .)
2. AccRun	Do a periodic action
3. AccCursor	Change cursor shape
4. AccMenu	Menu selection
5. AccUndo	The Undo editing command
6. AccCut	The Cut editing command
7. AccCopy	The Copy editing command
8. AccPaste	The Paste editing command
9. AccClear	The Clear editing command

Table 1: *DA actions allowed by Ctl*

```
PROCEDURE DoAppleChoice (Var  AppleMenu: MenuHandle;
                                          theItem    : INTEGER);
Var
   accName    : Str255;       {Name of desk accessory}
   accNumber  : INTEGER;      {Reference number of desk accessory}

Begin {DoAppleChoice}
Case theItem of
   AboutItem:          {Application's About... Item}
      DoAbout;
   otherwise  {Must be a DA...}
      Begin
      GetItem (AppleMenu, theItem, accName);
                                                {Get accessory name}
      accNumber := OpenDeskAcc (accName)    {Open desk accessory}
      End {otherwise}
 End {case}
End;          {DoAppleChoice}
```

Example 4: *The DoAppleChoice routine*

```
Procedure DoEditChoice (...., theitem : Integer);
   { Handle choice from Edit Menu }
   Begin                          { DoEditChoice}
       if Not SystemEdit (theItem-1)
      Then
          Case theItem of
             cutitem       : DoCut;
             copyitem    :  DoCopy;
             Pasteitem   : DoPaste;
       end; { Case}
End;       {DoEditChoice}
```

Example 5: *The DoEditChoice routine*

(continued from page 31)

by the linker with ID=1 and converting it to a *DRVR* resource. This is done by including the following lines in the resource file prior to compiling it with RMaker:

```
Type DRVR = PROC
   Deskacc,16
DeskaccFile
```

The name *DeskaccFile* is the linker's output file for the DA file, and *Deskacc* will be used as the DA name to appear under the Apple menu once the DA is installed in the system file. The resource ID then becomes the DA's driver reference number and is used by the operating system to implement the DA and is also used for owned resources.

Owned Resources

A range of 32 resource IDs has been reserved for each DA *DRVR* resource ID, so they are called owned resources. A special numbering convention is used to associate owned system resources with the resources they belong to. For any particular DA, this range is computed by adding $C000 and 32 times the driver reference number—for example, if the driver reference number is 16, then the range is –15,872 through –15,857.

When the DA is moved from its own file or a system file into a system file, all its resources for windows, menus, and so on must be transferred along with the code for the DA into the destination system file. If the destination system file already has a DA with the same *DRVR* resource ID, the Font/DA Mover will renumber it and all of its owned resources. Part of the DAMacMan resource file is shown in Listing One, beginning on page 48.

In summary, a DA program consists of at least three special procedures called *Open*, *Close*, and *Ctl*. The DA program may have other procedures as well, but it has no main body. You might think of the DA program as a module consisting of its own constants,g types, variables (Listing Two, page 48), and procedures (Listing Three, page 50). *Open* is called by *OpenDeskAcc* (from the running application); *Close* is called by *Ctl* (when the DA terminates itself) or *ExitToShell* (when the application terminates); and *Ctl* is called each time the application calls *SystemEvent*.

Listing Three shows these three procedures for DAMacMan, but keep in mind that these procedures must always exist for every DA even if they are tailored for some other purpose. In addition, because DAs are limited to 32K in size, the sophistication of a DA is restricted to miniature utility functions such as displaying the keyboard and so on. Implementing the MacMan database retrieval code was quite a challenge because of this limitation.

The Structure of DAMacMan

When DAMacMan is opened from the Apple menu, it looks in the system disk to see if the files it requires are present. If a file named Manual and another file named MMIndex are not present, DAMacMan will display an error dialog.

DAMacMan's related files are Manual, text from *Inside Macintosh* (the database); MMIndex, the index into the database; MMSize.int, a temporary file used to generate a version of DAMacMan; MMConfig, a DAMacMan software tool for generating versions and MMLock, which locks and unlocks files.

Manual, the database, consists of two parts: the MacMan distribution text and the MacMan information that contains all the procedures and functions defined in *Inside Macintosh*, organized as follows:

```
\ name
   category#
   body
```

where *name* is the name of the procedure or function, *category* is the section of *Inside Macintosh* in which it is defined, and *body* is the information about the procedure or function.

The index file, MMIndex, contains records sorted with respect to name, each record having the following form:

```
Record
   name : String[25];
   start : longint;
   length : Integer;
   man : Integer;
end;
```

where *name* is the name of the procedure or function, *start* is the starting position relative to the beginning of the manual, *length* is the length of the text that belongs to this procedure or function, and *man* is an integer representing the section or manager of *Inside Macintosh* where this function or procedure is defined.

MMLock is a MacMan tool whose job is to lock and unlock the files that are generated

The MMSize.int file contains the statement:

```
Const MaxRec =    ;
```

The blank will be filled in by the MacMan tool MMConfig, prior to compiling MacMan. The value of *MaxRec* is equal to the number of entries in Manual. The DAMacman main program is shown in Example 6, page 41.

MacMan Tools

DAMacMan is configured and maintained through the use of a set of tools that must be applied each time the database is changed. MMConfig is a tool for constructing the index file for the database and the Pascal compiler *include* file that contains the size of the database. In addition, the integrity of the database is maintained by locking the database files using the MMLock program, described later.

MMConfig reads the file Manual and writes the ordered file MMIndex with records of the form:

```
Record
   name : String [25];
```

```
  start : Longint;
  length : Integer;
  man : Integer;
end;
```

MMConfig performs the following steps:

1. It forms a record for each procedure or function in Manual.
2. It store a pointer to each of the records in an array of pointers.
3. It sorts the array with respect to name.
4. It creates the file MMIndex, to be read whenever MacMan is started up, and writes the sorted records to it.
5. It creates the file MMSize.int, which will be included during compilation of DAMacMan.
6. It locks the files Manual and MMSize.int so they cannot be opened by a user who is not supposed to have access to these files.
7. It stamps these files with the appropriate icons.

MMConfig is run to create MMIndex and MMSize.int after Manual has been edited the first time and whenever Manual is updated.

Finally, MMLock is a MacMan tool whose job is to lock and unlock the files that are generated by MMConfig.

Data Structures

As mentioned, after Manual has been constructed by entering all desired text into the database, MMConfig is run to generate the files MMIndex and MMSize.int. Then DAMacMan can be compiled and run. When DAMacMan is run, it loads the file MMIndex into a list of type *table*:

```
table : array [1. .MaxRec] of ptr1;
```

where *ptr1* is a pointer to a record similar to those stored in MMIndex and *MaxRec* is the number of functions or procedures in Manual. *MaxRec* is set by including the file MMSize.int.

Installing DAMacMan

DAMacMan is compiled into the resource file MacManFile. The Font/DA Mover utility tool is used to copy this file into a system file. After the DA has been installed in the system file of the start-up volume, it can be selected from the Apple menu.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, MACINTOSH, KAYPRO).

The complete source code and binary files for DAMacMan are also available on Macintosh disk (single-sided) from the authors.

Bibliography

Apple Computer Inc. *Inside Macintosh*. 4 vols. Reading, Mass.: Addison-Wesley, 1985–1986.

Chernicoff, Stephen. *Macintosh Revealed*, Volumes I and II. Hasbrouck Heights, N.J.: Hayden/Apple Press, 1985.

Reingold, E.; and Wilfred, H. *Data Structures*. Boston: Little, Brown, 1983.

Takatsuka, J.; Huxham, F; and Burnard, D. *Using the Macintosh Toolbox with C*. Berkeley, Calif.: Sybex, 1986.

TML Systems. *MacLanguage Series Pascal User's Guide and Reference Manual*. Jacksonville, Fla.: TML Systems, 1985.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

Wootton, Alan. "Resource Utility DA with TML." *MacTutor* (November 1985).

DDJ

(Listings begin on page 48.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

MAC DATA BASE

```
PROGRAM  DAMacMan;
{ DAMacMan is an online inside macintosh, running as a desk accessory

  Pascal source:     DAMacMan.pas     Main Program
  Uses    :

           MMSize.Int        The Manual Size, Generated
                                                by MMConfig Tool
           DAMacMan.int    The Interface include file
           DAOthers.imp      Our own procedures and functions
           DAThree.imp        The open, Ctl, Close procedures
           DAIndex           Index To Database Entries
         Manual              Database from Inside Macintosh
  Resources  :       DAMacMan.R

  Creation Date:  July 1St, 1986
  Author     :          Abdullah  Al-Dhelaan
                     (TGL Software Development Group )
                     Oregon State University
}
{ The next four include files below are Interface to Toolbox ...}
{$I MemTypes.ipas   }
{$I QuickDraw.ipas   }
{$I OSIntf.ipas          }
{$I ToolIntf.ipas       }

{$I MMSize.Int          }     (The Manual Size, Generated
                                                by MMConfig Tool }
{$I DAMacMan.int      }   { The Interface include file    }
{$I DAOthers.imp        }    { Our own procedures and functions }
{$I DAThree.imp          }    {The open, Ctl, Close procedures }

{-------------------------Main Program----------------------------}
BEGIN
        { Desk Accessory,  There should be no main program }
END.
```

Example 6: *The DAMacMan main program*

Listing Twelve *(Text in December.)*

EXTERNS.H

```
/*
   This include file contains all of the external symbol definitions
   for modules that refer to external data.
*/

#ifndef  GLOBALS

extern   unsigned int   load_segment ;
extern   SEG_DESCRIPTOR *seg_list ;

extern   char  config_fname[FILENAME], abs_fname[FILENAME] ;
extern   char  module_name[FILENAME], locate_fname[FILENAME] ;
extern   char  print_fname[FILENAME], exe_fname[FILENAME] ;
extern   char  map_fname[FILENAME], tmp_fname[FILENAME] ;

extern   char  command_line[COMMAND_LINE] ;

extern   int   tmp_file ;
extern   int   abs_file ;
extern   int   exe_file ;

extern   FILE  *map_file ;
extern   FILE  *print_file ;
extern   FILE  *config_file ;

extern   BOOLEAN  boot_rec, help, config, user_abort, hex_name ;

#endif
```

End Listing Twelve

Listing Thirteen

GLOBALS.H

```
/*
   This module contains all of the global data declarations
*/

#ifndef  GLOBALS

unsigned int   load_segment ;
SEG_DESCRIPTOR *seg_list ;

char  config_fname[FILENAME], abs_fname[FILENAME] ;
char  module_name[FILENAME], locate_fname[FILENAME] ;
char  print_fname[FILENAME], exe_fname[FILENAME] ;
char  map_fname[FILENAME], tmp_fname[FILENAME] ;

char  command_line[COMMAND_LINE] ;

int   tmp_file ;
int   abs_file ;
int   exe_file ;

FILE  *map_file ;
FILE  *print_file ;
FILE  *config_file ;

BOOLEAN  boot_rec = FALSE, help = FALSE, config = FALSE ;
BOOLEAN  hex_name = FALSE, user_abort = FALSE ;

#define  GLOBALS

#endif
```

End Listing Thirteen

Listing Fourteen

LOC.H

```
#define  MAX_TOKEN      80
#define  MAX_LINE       80
#define  FILENAME       80
#define  COMMAND_LINE   256

#define  PAGE_SIZE      512

#define  ERROR    0     /* Error condition exists */
#define  OK       1     /* No error(s) detected */

#define  T_EOL    -2    /* Reached the end of a line */
#define  T_ERROR  -1    /* Detected an error condition */
#define  T_OK     0     /* Everything fine */
#define  T_OCT    1     /* Octal format integer */
#define  T_DEC    2     /* Decimal format integer */
#define  T_HEX    3     /* Hexidecimal format integer */
#define  T_WORD   4     /* Symbol character string */
#define  T_OP     5     /* Operator character string */

typedef  enum  { FALSE = 0, TRUE }  BOOLEAN ;

#define  low_byte(x)     ((x) & 0xff)
#define  high_byte(x)    (low_byte((x) >> 8))
#define  dim(x)          (sizeof(x)/sizeof(x[0]))

typedef  struct {
      unsigned int   signature ;
      unsigned int   length ;
      unsigned int   pages ;
      unsigned int   reloc_items ;
      unsigned int   header_size ;
      unsigned int   min_para ;
      unsigned int   max_para ;
      unsigned int   stack_seg_disp ;
      unsigned int   initial_sp ;
      unsigned int   checksum ;
      unsigned int   initial_pc ;
      unsigned int   code_seg_disp ;
      unsigned int   first_reloc_item ;
      unsigned int   overlay_num ;
   } EXE_HEADER ;


typedef  struct sym_list  {
            struct sym_list *next ;
            char  name[32] ;
            unsigned int   value ;
            unsigned char  type ;
         } SYMBOL_LIST ;

typedef  struct  seg_descriptor  {
            struct seg_descriptor   *next ;
            long  position ;
            char  name[32] ;
            char  class[32] ;
            unsigned int   vseg ;
            unsigned int   pseg ;
            unsigned int   offset ;
            unsigned int   len ;
            int   inited ;
            int   romable ;
            unsigned int   symbols ;
            SYMBOL_LIST *symbol_list ;
         } SEG_DESCRIPTOR ;

char  *load_exe_file() ;
SEG_DESCRIPTOR *build_seg_list() ;

int   get_ch() ;
int   unget_ch() ;
char  *get_cmd() ;
char  *get_line() ;

int get_token() ;

char  *get_mem(unsigned long) ;
void  free_mem(char *) ;

char  *process_command_line() ;
void  read_symbol_table(SEG_DESCRIPTOR *) ;

void  output_hex_OMF(int, SEG_DESCRIPTOR *, unsigned char *) ;
void  open_file_system() ;
void  close_file_system() ;
void  break_handler() ;

#define  DATA_RECORD    0
#define  EOF_RECORD     1
#define  ADDR_RECORD    2
#define  START_RECORD   3

void  write_START_record(int, unsigned char *) ;
void  write_ADDR_record(int, unsigned int ) ;
void  write_DATA_record(int, unsigned int, unsigned char *,
                        unsigned int ) ;
void  write_EOF_record(int) ;
void  output_hex_record(int, unsigned char, unsigned int,
                        unsigned char *, unsigned char) ;;
```

End Listing Fourteen

Listing Fifteen

```
            page    60, 132
            name    startup
            title   LOCATE Example ROM System Startup Code
            subttl  Turbo C 1.0 Version

;
;       ROM System Startup Code for Turbo C 1.0
;       Copyright (C) 1987 by Rick Naro.  All rights reserved.
;

;
;       Segment and group declarations
;

_text   segment byte    public  'CODE'
_text   ends
_etext  segment para    public  'CODEEND'
_etext  ends
_data   segment para    public  'DATA'
_data   ends
_bss    segment para    public  'BSS'
_bss    ends
_bssend segment byte    public  'BSSEND'
_bssend ends
_stack  segment para    stack   'STACK'
_stack  ends

dgroup  group   _data, _bss, _bssend

assume  cs:_text, ds:dgroup, ss:_stack

;
;       This version of the startup code expects 32-bit code pointers.
;       (medium or large memory models)
;

extrn   _main : far     ; Change to near for small/compact memory model

_text   segment

public  start

start   proc    far

        cli                             ; Disable interrupts

;
;       Initialize the stack segment and pointer
;

```

Listing Fifteen *(Listing continued.)*

```
51          mov     ax, _stack              ; Get the stack segment value
52          mov     ss, ax                  ; Put in SS
53          mov     sp, offset tos          ; Load the TOS in SP
54
55  ;
56  ;       Set up the segment registers for the initialization of DGROUP.
57  ;       ES is initialized to DGROUP and DS is initialized to the copy of
58  ;       DGROUP in ROM.
59  ;
60
61          mov     ax, dgroup              ; Get the segment for DGROUP
62          mov     es, ax                  ; Install in ES
63
64          mov     ax, _etext              ; Get the _etext segment
65          inc     ax                      ; Adjust for the size of _etext
66          mov     ds, ax                  ; Install in DS
67
68  ;
69  ;       Copy the copy of the initialized data segment _DATA from its
70  ;       position in ROM (just after the CODE class) to the real _DATA
71  ;       segment in RAM.  The size of the _DATA segment is computed by
72  ;       subtracting the start of _DATA (the label idata) from the start
73  ;       of the next segment (the label bdata).
74  ;
75
76          mov     si, offset DGROUP:idata ; Starting offset of _DATA
77          mov     di, si                  ; Same offset, different segment
78          mov     cx, offset DGROUP:bdata ; Get the end offset
79          sub     cx, di                  ; Subtract the two for a length
80          rep     movsb                   ; Copy initialized data
81
82          push    es                      ; Get the value of DGROUP
83          pop     ds                      ; Now it is in DS
84
85  ;
86  ;       C expects that the BSS segment is cleared when the program is
87  ;       started.  Zero out the BSS segment within DGROUP by computing
88  ;       the size using the label bdata and edata labels
89  ;
90
91          xor     al, al                  ; Use a zero fill pattern
92          mov     di, offset DGROUP:bdata ; Get the starting offset
93          mov     cx, offset DGROUP:edata ; Get the ending offset
94          sub     cx, di                  ; Subtract the two for a length
95          rep     stosb                   ; Zero out the BSS
96
97  ;
98  ;       Call main and hopefully never return ...
99  ;
100
101         sti                             ; Re-enable interrupts
102         call    _main                   ; Call main()
103         ret
104 ;
105 ;       Handle a return by restarting the entire process
106 ;
107
108         jmp     start                   ; Start all over
109
110 start   endp
111
112 _text   ends
113
114         page
115 _etext  segment
116 public  tend
117         db      16 dup (?)              ; Force alignment of the label
118                                         ; tend with the next paragraph
119 tend    label   byte                    ; Mark the end of the segment
120                                         ; Do NOT change this segment
121 _etext  ends
122
123 _data   segment
124 public  idata
125 idata   label   byte                    ; Start of the _DATA segment
126 _data   ends
127
128 _bss    segment
129 public  bdata
130 bdata   label   byte                    ; Start of the _BSS (and the
131                                         ; end of the _DATA segment
132 _bss    ends
133
134 _bssend segment
135 public  edata
136 edata   label   byte                    ; Mark the end of the BSS
137 _bssend ends
138
139 _stack  segment
140 public  tos
141         dw      256 dup (?)             ; Declare the stack size
142 tos     label   byte                    ; Define the top of stack
143 _stack  ends
144
145         end     start
```

End Listings

MAC DATA BASE

Listing One *(Text begins on page 24.)*

```
*           DAMacMan.r                  Resource
*           July 3, 1986                Last modified
*
* Resource file for "DAMacMan.pas" for use with MacLanguage
* series Pascal Compiler (TML Pascal)
*
*
* all resources must be between -15872 and -15841 inclusive
* if they are to travel  with DRVR number 16

MacManFile
DFILDMOV

TYPE DRVR = PROC
  MacMan,16
DAMacMan

* this DLOG is the main window.
* the StatText items are used only for their rectangles.

Type WIND
  ,-15872
Untitled
50 40 300 510
InVisible GoAway
0
0

* menu used by the DA

type MENU
  ,-15872
Macman      ;; menu title
about Macman
(-
Find By Name
View By Category

* This second dialog is for about menu item
* the button and icon do nothing. Just decoration

Type ALRT
  ,-15872
50 50 330 430
-15872
4444

  ,-15871
50 50 300 480
-15871
4444

  ,-15870
60 80 126 430
-15870
4444
```

End Listing One

Listing Two

```
{
            DAMacMan.int                Interface
            July 3, 1986                    Last modified

   An interface file that Contains all the needed types and vars

}

Const

  accEvent   = 64;
  accRun      = 65;
  accCursor = 66;
  accMenu   = 67;
  accUndo     = 68;
  accCut      = 70;
  accCopy    = 71;
  accPaste   = 72;
  accClear   = 73;

  ManFile     = 'Manual';    {Name of Database File}
  Vnum        = 0;  {default drive}

 Type
    Str25  = String [25];
    Lptr   = ^LongInt;
    Item   =  Record                        { An entity record}
                 name       : Str25;            { Proc/Func. Name }
                   start          : LongInt;         { Starting pos. on the
                                                       Manual }

                 length     : Integer;       { Length of Proc./Func. Text }
                 man        : Integer;       { Category # }
              END;

 ptrl  =  ^ Item;
 myarray  = ARRAY [1..MaxRec] of Ptrl;  { The main array }

{This is the data. A handle to it will be stored in the dctl storage field
  of the DCltEntry Record  }

 GlobalsH = ^GlobalsP;
 GlobalsP = ^GlobalsRec;
 GlobalsRec = Record { DAMacMan Database Info }
                      hScroll   : ControlHandle;
                                              {Horizontal scroll for window.}
                      vScroll   : ControlHandle;{Vertical scroll for window}
                      pRect     : Rect;       {Rectangle within window to see}
                    tRect     : Rect;
                    hTE       : TEHandle;     {Text is here...}
                      table     : myarray;      {Index to entries}
                  noIndex   : boolean;          {Error if no Index on Disk}
                     noman     : boolean;       {Error if no Database on Disk}
                   END;

{ This is used to store system info about the state of the driver. It will
  be passed to us on all calls from system. This Record is definded in the
  interface (TML files) above (as DCtlEntry) . it is not strictly necessary to
  redefine it here. But doing so,  will enable me to use dctlStorage^^ to
  refer to GlobalsRec without coercing Types }

 MyDeviceEntry = Record
     DctlDriver  : HAndle;    { Pointer to driver }
     DctlFlags    : Integer;  { Flags }
     DctlQueue   : Integer;   { Low-order byte, drivers version number }
     DctlQhead   : Lptr;        { Pointer to first entry in drivers I/O queue }
     DctlQTail     : Lptr;        { pointer to last entry in drivers I/O queue }
     DctlPosition: LongInt;  { Byte position }
     DctlStorage : GlobalsH; { Handle to RAM driver's private storage }
     DctlRefNum : Integer;  { Driver's reference number }
     DctlCurTicks: LongInt;  { Used internally by Device Manager }
     DctlWindow : Grafptr;  { Pointer to driver's window Record }
     DctlDelay    : Integer;  { number of ticks between periodic actions }
     DctlEmask    : Integer;   { Desk accessory event mask }
 DctlMenu     : Integer;   { Menu ID of menu associated with driver }
  END;
{ No VARiables for main program are allowed }
```

End Listing Two

Listing Three

```
{
            DAThree.imp             Implementatioon
```

```
            July 3, 1986                Last modified

      THE FOLLOWING PROCEDURES SHOULD BE IN EVERY DESK ACCESSORY
      AND THEY ARE CALLED BY THE SYSTEM



 PROCEDURE open    ( VAR Device : MyDeviceEntry;
                     VAR block  :  ParamblockRec);
 { Open Makes a window and sets-up our private storage.
   we may get an open call even after we are already open }

CONST
       InfoLength = 700;   {The number of chars. fro the distribution text}
   VAR
     Wpk       : WindowPeek;
     Dtyp      : Integer;
     ID        : Integer;
     Dhan      : Handle;
     tmpPtr    : ptr;
     dummy     : GlobalsRec;
     Watch     : CursHandle;  {Handle to wristwatch cursor}

   BEGIN

   {Use ID for all Resource access}
   ID := $C000 - 32 * (1 + Device.dctlRefNum);
   Device.dctlMenu := ID;
    With Device do
    if DctlWindow = nil
    THEN BEGIN
       { Create a hole in the heap.  It is good practice to keep
            Window records off of the bottom of the Application Heap. }
         DctlStorage := pointer(NewHandle(SizeOf(dummy)));
         TmpPtr := NewPtr($1000);
         Hlock(Handle(DctlStorage));
         With DctlStorage^^ do
          BEGIN {initialize our storage }

          noindex := false;  {Index found}
             noman   := false;   {Manual found}
             Watch := GetCursor (WatchCursor);
             SetCursor (Watch^^);                    {Indicate delay }
           CreateWindow (Device,ID);
           { post information }
          DoOpen (Device,'MacMan Distribution',0,InfoLength);
              If not noman
              Then LoadData(device); {Load the array}
          initcursor;
       END;  { of with storage }
         {Deallocate our temporary pointer }
       Hunlock(Handle(DctlStorage));
        DisposPtr(TmpPtr);
     END; { of if }
 END; { of open }

{-------------------------------------------------------------------------}
PROCEDURE close  (   VAR Device : MyDeviceEntry;
                     VAR block    : paramBlockRec);
 BEGIN
    deactivate(Device);       { remove menu }
    with Device do
    BEGIN
      disposHandle(Handle(DctlStorage));       { kill data }
      disposeWindow(DctlWindow);               { erase window }
      DctlWindow := nil;
   END; {of with }
 END; { of close }

{-------------------------------------------------------------------------}

 PROCEDURE ctl (     VAR Device : MyDeviceEntry;
                     VAR block : ParamBlockRec) ;
{ Here is the main entry point for system calls. The permanent bolck tell us
  what the nature of the call is. }
  VAR
      mousept      : point;
      wpnt         : Grafptr;
      item         : Integer;
      ibeam        : cursHandle;
      ignore       : Integer;
      longignore   : LongInt;

   BEGIN
      setport(Device.DctlWindow);
      Hlock(Handle(Device.DctlStorage));
      with Device, DctlStorage^^,block do
       BEGIN
          TEidle(hTE);
          CASE csCode of
              accevent    :  Event(Device, block);
              accCursor :
                BEGIN
                   ibeam := GetCursor(ibeamcursor);
                   GetMouse(mousePt);
                   If (PtInRect(mousePt,pRect))
                     THEN SetCursor(iBeam^^)
                     ELSE InitCursor;
                END;
             accmenu    :     { CASE out menu item number }
               BEGIN
                  Initcursor;
                  CASE csParam[1] of
                        1 : Doabout (Device.dctlmenu);   {about... }
                      3 : IF (not noindex) THEN  DoFind  (Device,'');
                      4 : IF (not noindex) THEN  DoView  (Device)  ;
                    END; { of CASE menu }
                  HiliteMenu(0);
              END; { of menu CASE }
            accCopy :
             BEGIN
                      longignore := ZeroScrap;  {Init. Scrap}
                TECopy(hTE);           {Copy text from hte to TextEdit scrap}
                ignore := TEToScrap;  {Copy  TextEdit scrap to desk scrap }
                ignore := UnloadScrap; {Copy desk scrap to file scrap}
             END;
           Otherwise ;
        END; { CASE of ... }
       END; { of with block }
     Hunlock(Handle(Device.DctlStorage));
   END; { of control  PROCEDURE }
```

TO THE MACS

Listing One *(Text begins on page 90.)*

```
/*------------------------------ file information ------------------------------*/


/*
        custom controls demo.c

                c source code file for a minimal Mac program that demonstrates
                        controls drawn with a custom CDEF resource

                the custom CDEF resource that's demonstrated provides 16 button
variations
                the buttons …
                                * … live in a rectangular space
                                * … can be outlined, shadowed, or bare
                                * … can contain text in any font-style-size, an icon, or
a picture
                                * … can indicate highlighting via inversion or a change
of content

        edited and compiled with Lightspeed C 2.13

        written and ©1987 by Stan Krute. all rights reserved. no part of this file,
        or the object code it leads to, may be reproduced, in any form or by any means,
        without        the express written permission of the author and copyright
holder.

        timestamp:              3:49 pm PST                     November 16, 1987
        spacestamp:             18617 Camp Creek Road        Hornbrook, California
96044

        this file looks good in 9 point Courier, LSC tabs set to 3
*/


/*-------------------------------- include files -------------------------------*/

/* definitions for Mac OS managers used herein */
#include        "ControlMgr.h"
#include        "DialogMgr.h"
#include        "EventMgr.h"
#include        "FontMgr.h"
#include        "MenuMgr.h"
#include        "Quickdraw.h"
#include        "StdFilePkg.h"

/* our stuff */
#include        "custom controls demo.h"            /*   private definitions for this
file  */


/*------------------------------ main program block ----------------------------*/

void    main()

                {
                /* local variable */
                int             theItem ;

                /* initialize Mac OS managers  */
                initializeManagers() ;

                /* see what the world is like */
                studyAndSetEnvironment () ;

                /* set up and draw a (dummy) title menu */
                InsertMenu( GetMenu(titleMenuID), append ) ;
                DrawMenuBar() ;

                /* set up and draw a modal dialog window */
                getThatDialogCookin () ;

                /* initalize our doneness indicator */
                finished = false ;

                /*  run the main event loop   */
                do

                        {
                        ModalDialog (noFilterProcedure, &theItem) ;
                        dealWithDialogItem (theItem) ;
                        }
                while

                        ( ! finished ) ;

                /*  leave neatly when done */
                DisposDialog (ourDialog) ;                  /* bye bye to dialog */
                ExitToShell() ;                                         /* bye bye to
program */
                }


/*---------------------------- initializeManagers ----------------------------*/

/*   initialize the heap, cursor, and Mac Operating System managers    */

void    initializeManagers()

                {
                /* local variable */
                Handle          someDay ;

                /* get some space */
                MoreMasters() ;
                /* get some master pointers */
                if (someDay = NewHandle(humungousBlock))    /* grow a maximal heap by */
                        DisposHandle (someDay) ;            /* asking for the future */
                                                            /* get those managers going */

                InitGraf(&thePort) ;                        /* set up  Quickdraw */

                InitFonts();                                /* set up the Font Manager */

                InitWindows();
                /* set up the Window Manager */
                InitMenus();                               /* sot up the Menu Manager */

                TEInit();
                /* set up Text Edit */
                InitDialogs (noResumeProcedure) ;           /* set up the Dialog Manager */
```

```
        /* final adjustments */
        FlushEvents (everyEvent, dontStop ) ;       /* clear the event queue */
        InitCursor();
        /* turn the cursor on */
        }

/*------------------------------ studyAndSetEnvironment --------------------------*/

/*   check out screens, machines, ROMs, et al */

void    studyAndSetEnvironment ()

        {
        /* check out the screen */
        screenRect = screenBits.bounds ;
        screenHeight = screenRect.bottom - screenRect.top ;
        screenWidth = screenRect.right - screenRect.left ;

        /* determine height of the menu bar */
        if      ( ROM85 & 0x8000 )
                menuBarHeight = stdMBarHeight ;      /* for 64K ROMs       */
        else
                menuBarHeight = MBarHeight ;         /* for newer  ROMs    */

        }


/*------------------------------- getThatDialogCookin -------------------------*/

/* set up and draw our main modal dialog window */

void    getThatDialogCookin ()

        {
        /* local variables */
        Point                   tempPoint ;
        Rect                    scratch ;
        ControlHandle  theButton ;

        /* get the dialog window */
        ourDialog = GetNewDialog (ourDialogID, storeInHeap, inFront) ;

        /* adjust its position  */
        MoveWindow ( ourDialog,
                    (tempPoint = figureCenteradRectTLC  (&(*ourDialog).portRect)).h,
                     tempPoint.v, inFront ) ;


        /* make dialog window the current grafPort so we can change its font */
        SetPort (ourDialog) ;

        /* change its font to Geneva 12 */
        TextFont (geneva) ;
        TextSize (12) ;

        /* show the dialog */
        ShowWindow (ourDialog ) ;
        }


/*------------------------------- dealWithDialogItem -------------------------*/

 /* deal with the hit item */

void    dealWithDialogItem (theItem)
        int             theItem ;

        {
/* local constants */
#define             oolSize             6

/* local variables */
static short  onOffList[oolSize] = {       orwellItem, hupCoupleItem,

        ronItem, saveAsItem,

        pinheadItem, duplicateItem} ;
/* case out on the item */
switch (theItem)
       {
       case quitItem:
              finished = true ;
              break ;
       case orwellItem:
              doOrwellItem () ;
              break ;
       case snapshotItem:
              doSnapshotItem () ;
              break ;
       case mushroomItem:
              doMushroomItem () ;
              break ;
       case openItem:
              doOpenItem () ;
              break ;
       case saveAsItem:
              doSaveAsItem () ;
              break ;
       case flipItem:
              doFlipItem () ;
              break ;
       case someOffItem:
              doSomeOffItem (onOffList, oolSize) ;
              break ;
       case someOnItem:
              doSomeOnItem (onOffList, oolSize) ;
              break ;
       case copyrightItem:
              doCopyrightItem () ;
              break ;
       default:

                       break ;
                }

          /* remove local constants */
          #undef        oolSize
          }


 /*------------------------------- doOrwellItem -------------------------------*/

 /* deal with a click of the orwellItem  button */

 void    doOrwellItem ()

          {
          /* local constants */
          #define                       cyclesDesired         4
          #define                       delayTicksOne         20
          #define                       delayTicksTwo         10

          /* local variables */
          Rect                          scratch ;
          ControlHandle  theItemHandle ;
          short                         cycleCounter ;
          ControlHandle  ronItemHandle ;

          /* get a handle to the button */
          GetDItem ( ourDialog, orwellItem, &scratch, &theItemHandle, &scratch) ;

          /* hilite the button */
          HiliteControl (theItemHandle, hilitedHS ) ;

          /* get a handle to the ronItem button */
          GetDItem ( ourDialog, ronItem, &scratch, &ronItemHandle, &scratch) ;

          /* run several fade cycles on the ronItem button  */
          for ( cycleCounter = 0; cycleCounter < cyclesDesired; cycleCounter++)
                 {
                 /* fade out */
                 HiliteControl (ronItemHandle, inactiveHS ) ;

                 /* wait a while */
                 Delay (delayTicksOne, &scratch) ;

                 /* back into view */
                 HiliteControl (ronItemHandle, activeHS ) ;

                /* wait a while */
                Delay (delayTicksTwo, &scratch) ;
                }

          /* unhilite the button */
          HiliteControl (theItemHandle, activeHS ) ;

          /* remove local constants */
          #undef        cyclesDesired
          #undef        delayTicksOne
          #undef        delayTicksTwo
          }


/*------------------------------- doSnapshotItem -----------------------------*/

/* deal with a click of the snapshotItem button */

void    doSnapshotItem ()

          {
          /* local variables */
          ControlHandle  theItemHandle ;
          Rect                          scratch ;
          GrafPtr                       entryGrafPort ;

          /* get a handle to the button */
          GetDItem ( ourDialog, snapshotItem, &scratch, &theItemHandle, &scratch)
;

          /* hilite the button */
          HiliteControl (theItemHandle, hilitedHS ) ;

          /* save a pointer to the grafPort */
          GetPort(&entryGrafPort) ;

          /* this lets me take some snapshots */
          /* has no effect unless you have a desk accessory named Camera */

          OpenDeskAcc ("\007\000Camera") ;

          /* restore the grafPort */
          SetPort(entryGrafPort) ;

          /* unhilite the button */
          HiliteControl (theItemHandle, activeHS ) ;
          }


/*------------------------------- doMushroomItem -----------------------------*/

/* deal with a click of the mushroomItem  button */

void    doMushroomItem ()

          {
          /* local constants */
          #define                       cyclesDesired         4
          #define                       delayTicks                    30

          /* local variables */
          Rect                          scratch ;
          short                         cycleCounter ;
          ControlHandle  itemHandleOne ;
          ControlHandle  itemHandleTwo ;

          /* get a handle to the button */
          GetDItem ( ourDialog, mushroomItem, &scratch, &itemHandleOne, &scratch)
;

          /* hilite the button */
          HiliteControl (itemHandleOne, hilitedHS ) ;

          /* get a handle to the bumperStickersItem button */
          GetDItem ( ourDialog, bumperStickersItem, &scratch,
                                        &itemHandleTwo, &scratch) ;

          /* run several fade cycles on the bumperStickersItem button  */
          for ( cycleCounter = 0; cycleCounter < cyclesDesired; cycleCounter++)
                 {
                 /* fade out */
                 HiliteControl (itemHandleTwo, hilitedHS ) ;

                 /* wait a while */
                 Delay (delayTicks, &scratch) ;

                 /* back into view */
                 HiliteControl (itemHandleTwo, activeHS ) ;

                 /* wait a while */
                 Delay (delayTicks, &scratch) ;
                 }

          /* unhilite the button */
          HiliteControl (itemHandleOne, activeHS ) ;

          /* remove local constants */
          #undef        cyclesDesired
          #undef        delayTicks
          }
```

```
/*---------------------------------- doOpenItem ------------------------------*/

/* deal with a click of the openItem button */

void    doOpenItem ()

        {
        /* local variables */
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        DialogTHndl             theDLOGHandle ;
        SFReply                 dummyReply ;

        /* get a handle to the button */
        GetDItem ( ourDialog, openItem, &scratch, &theItemHandle, &scratch) ;

        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* run the standard file open dialog */
        theDLOGHandle = (DialogTHndl) GetResource ('DLOG', getDlgID) ;
        SFGetFile (figureCenteredRectTLC (&(**theDLOGHandle).boundsRect),
                                nil, nil, allTypes, nil, nil, &dummyReply ) ;


        /* unhilite the button */
        HiliteControl (theItemHandle, activeHS ) ;
        } ;


/*---------------------------------- doSaveAsItem ----------------------------------*/

/* deal with a click of the saveAsItem button */

void    doSaveAsItem ()

        {
        /* local variables */
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        DialogTHndl             theDLOGHandle ;
        SFReply                 dummyReply ;

        /* get a handle to the button */
        GetDItem ( ourDialog, saveAsItem, &scratch, &theItemHandle, &scratch) ;

        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* run the standard file open dialog */
        theDLOGHandle = (DialogTHndl) GetResource ('DLOG', putDlgID) ;
        SFPutFile (figureCenteredRectTLC (&(**theDLOGHandle).boundsRect),
                        "\015Save file as:", "\021Current File Name",
                        nil, &dummyReply ) ;


        /* unhilite the button */
        HiliteControl (theItemHandle, activeHS ) ;
        }


/*---------------------------------- doFlipItem ----------------------------------*/

/* deal with a click of the flipItem button */

void    doFlipItem ()

        {
        /* local constants */
        #define         numButtons          9
        #define         cyclesDesired  4
        #define         delayTicks          15

        /* local variables */
        static short  flipList [numButtons] =
                        {       hupCoupleItem,          mouthOpensItem,
                        trashItem,                      melancholyItem,
                        eeekShrinkItem,                 mushroomItem,
                        duplicateItem,                  orwellItem,
                        bumperStickersItem
        } ;
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        short                   cycleCounter ;
        short                   index ;
        ControlHandle  tempItemHandle ;
        /* get a handle to the button */
        GetDItem ( ourDialog, flipItem, &scratch, &theItemHandle, &scratch) ;

        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* run several animation cycles on a group of content-changing buttons
*/
        for ( cycleCounter = 0; cycleCounter < cyclesDesired; cycleCounter++)
                {
                /* hilite all the buttons in the group  */
                for (index = 0 ; index < numButtons ; index++)
                        {
                        GetDItem ( ourDialog, flipList[index], &scratch,
                                        &tempItemHandle, &scratch) ;
                        HiliteControl (tempItemHandle, hilitedHS ) ;
                        }

                /* wait a while */
                Delay (delayTicks, &scratch) ;

                /* unhilite all the buttons in the group */
                for (index = 0 ; index < numButtons ; index++)
                        {
                        GetDItem ( ourDialog, flipList[index], &scratch,
                                        &tempItemHandle, &scratch) ;
                        HiliteControl (tempItemHandle, activeHS ) ;
                        }

                /* wait a while */
                Delay (delayTicks, &scratch) ;
                }

        /* unhilite the button */
        HiliteControl (theItemHandle, activeHS ) ;

        /* remove local constants */
        #undef          numButtons
        #undef          cyclesDesired
        #undef          delayTicks
        }


/*---------------------------------- doSomeOffItem ----------------------------------*/

/* deal with a click of the someOffItem button  */

void    doSomeOffItem (theOffList, listSize)
        short           theOffList[] ;
        short           listSize ;

        {
        /* local variables */
        short                   index ;
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        ControlHandle  tempItemHandle ;

        /* get a handle to the button  */
        GetDItem ( ourDialog, someOffItem, &scratch, &theItemHandle, &scratch) ;

        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* for each item in the list */
        for (index = 0 ; index < listSize ; index++)
                {
                /* get a handle to the item */
                GetDItem ( ourDialog, theOffList[index], &scratch,
                                &tempItemHandle, &scratch) ;

                /* inactivate the item */
                HiliteControl (tempItemHandle, inactiveHS ) ;
                }

        /* unhilite the someOffItem button  */
        HiliteControl (theItemHandle, activeHS ) ;
        }


/*---------------------------------- doSomeOnItem ----------------------------------*/

/* deal with a hit of the someOnItem button  */

void    doSomeOnItem (theOnList, listSize)
        short           theOnList[] ;
        short           listSize ;

        {
        /* local variables */
        short                   index ;
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        ControlHandle  tempItemHandle ;
        /* get a handle to the button  */
        GetDItem ( ourDialog, someOnItem, &scratch, &theItemHandle, &scratch) ;

        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* for each item in the list */
        for (index = 0 ; index < listSize ; index++)
                {
                /* get a handle to the item */
                GetDItem ( ourDialog, theOnList[index], &scratch,
                                &tempItemHandle, &scratch) ;

                /* activate the item */
                HiliteControl (tempItemHandle, activeHS ) ;
                }

        /* unhilite the someOnItem button  */
        HiliteControl (theItemHandle, activeHS ) ;
        }


/*----------------------------------doCopyrightItem ----------------------------------*/

/* deal with a hit on the copyrightItem button */

void    doCopyrightItem ()

        {
        /* local variables */
        Rect                    scratch ;
        ControlHandle  theItemHandle ;
        DialogPtr               copyrightDlog ;
        Point                   tempPoint ;

        /* get a handle to the button */
        GetDItem ( ourDialog, copyrightItem, &scratch, &theItemHandle, &scratch)


        /* hilite the button */
        HiliteControl (theItemHandle, hilitedHS ) ;

        /* pull in the copyright notice modal dialog */
        copyrightDlog = GetNewDialog (copyrightDlogID, storeInHeap, inFront) ;

        /* center it on the screen */
        MoveWindow ( copyrightDlog,
                        (tempPoint = figureCenteredRectTLC
(&(*copyrightDlog).portRect)).h,
                        tempPoint.v, inFront ) ;

        /* show the copyright notice */
        ShowWindow (copyrightDlog ) ;

        /* wait until the user clicks the mouse in the dialog */
        ModalDialog (noFilterProcedure, &scratch) ;

        /* get rid of the dialog */
        DisposDialog (copyrightDlog) ;

        /* unhilite the button */
        HiliteControl (theItemHandle, activeHS ) ;
        }


/*---------------------------- figureCenteredRectTLC ----------------------------*/

/* given a rectangle, returns the top left corner  position that will
        center the rectangle inside screen area that's below the menu bar */

Point   figureCenteredRectTLC (theRect)
```

```
            Rect    *theRect ;

            {
             /* local variable */
            Point   theResult ;

             /* figure the vertical position */
             theResult.v = menuBarHeight + ((screenHeight - menuBarHeight) -
                                                          (theRect-
>bottom - theRect->top) )  /  2 ;

             /* figure the horizontal position */
             theResult.h = ( screenWidth - (theRect->right - theRect->left) )   /
2 ;
   /* done, so return the point */
   return (theResult) ;
  }
```

Listing Two

```
/*------------------------------ file information ------------------------------*/

/*
        custom controls demo.h

        private definitions for custom controls demo.c

        edited and compiled with Lightspeed C 2.13

        ©1987 by Stan Krute    --    all rights reserved

        timestamp:              5:56 pm PST                     November 16, 1987
        spacestamp:             18617 Camp Creek Road    Hornbrook, California 96044


        this file looks good in 9 point Courier, LSC tabs set to 3
*/


/*-------------------------------- constants ---------------------------------*/

/* booleans */
#define                 true                            1
#define                 false                           0

/* parameters */
#define                 humungousBlock                  0x8FFFFFFF
#define                 noResumeProcedure               0
#define                 dontStop                        0
#define                 nil                             0
#define                 allTypes                        -1

/* control stuff */
#define                 activeHS                        0    /* three hilite states */
#define                 inactiveHS                      255
#define                 hilitedHS                       10


/* menu stuff */
#define                 stdMBarHeight 20
#define                 titleMenuID                     1
#define                 append                          0

/* dialog stuff */
#define         storeInHeap                             0
#define                 inFront                         -1
#define                 ourDialogID                     1
#define                 copyrightDlogID                 210
#define                 noFilterProcedure               0

#define                 quitItem                        1
#define                 orwellItem                      2
#define                 snapshotItem                    3
#define                 bumperStickersItem              4
#define                 mushroomItem                    5

#define                 hupCoupleItem                   6
#define                 openItem                        7
#define                 ronItem                         8
#define                 saveAsItem                      9
#define                 duplicateItem                   10

#define                 woozyItem                       11
#define                 trashItem                       12
#define                 flipItem                        13
#define                 mouthOpensItem                  14
#define                 copyItem                        15

#define                 melancholyItem                  16
#define                 pinheadItem                     17
#define                 someOffItem                     18
#define                 someOnItem                      19
#define                 eeekShrinkItem                  20
#define                 copyrightItem           21


/*-------------------------------- type definitions ----------------------------*/

typedef                 short   boolean;


/*------------------------------ function prototypes ---------------------------*/

void    main (void) ;
void    initializeManagers (void) ;
void    studyAndSetEnvironment (void) ;
void    getThatDialogCookin (void) ;
void    dealWithDialogItem (int    theItem) ;
void    doSnapshotItem (void) ;
void    doOrwellItem (void) ;
void    doMushroomItem (void) ;
void    doOpenItem (void) ;
void    doSaveAsItem (void) ;
void    doFlipItem (void) ;
void    doSomeOffItem (short  theOffList[], short   listSize);
void    doSomeOnItem (short  theOnList[], short   listSize);
void    doCopyrightItem (void) ;
Point   figureCenteredRectTLC (Rect   *theRect) ;


/*------------------------------ global variables ------------------------------*/

boolean                         finished ;                      /* indicates when the
program should end */
DialogPtr               ourDialog ;                     /* points to our main dialog */

int                             menuBarHeight ;         /* know your environment… */
int                             screenHeight ;
int                             screenWidth ;
Rect                            screenRect ;
```

End Listings

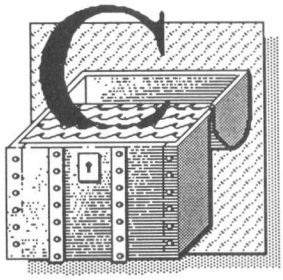
STRUCTURED PROGRAMMING

Listing One *(Listing continued, text begins on page 108.)*

```
Listing One.  Example for declaring various objects that are
              variations of stack structures
PROGRAM Test_Objects(input,output);

{ Simple example for objects using TML Pascal running on a Mac Plus }

TYPE RealStackReg = ARRAY [1..4] OF real;
     IntStackReg  = ARRAY [1..4] OF integer;

     { define common stack-related variables and routines }
     TStack = OBJECT
        height : integer;
       ErrorFlag : boolean;
        PROCEDURE TStack.IStack;
        PROCEDURE TStack.Inc(VAR i : integer);
        PROCEDURE TStack.Dec(VAR i : integer);
     END;

     { define a 4-register real-typed stack }
     TRealStack = OBJECT(TStack)
        Regs : RealStackReg;
       PROCEDURE Push(Item : real);
       FUNCTION Pop : real;
       PROCEDURE Add;
     END;

     { define a 4-register real-typed stack with automatic
     LastX register }
     THPStack = OBJECT(TRealStack)
        LastX : real;
       PROCEDURE Add; OVERRIDE;
     END;

     { define a 4-register integer-typed stack }
     TIntStack = OBJECT(TStack)
        Regs : IntStackReg;
       PROCEDURE Push(Item : integer);
       FUNCTION Pop : integer;
       PROCEDURE Add;
     END;


PROCEDURE TStack.IStack;
{ Initialize TStack objects by setting the Stack height to zero }
BEGIN
  height := 0
END; { Stack.IStack }
PROCEDURE TStack.Inc(VAR i : integer);

BEGIN
  i := i + 1;
END;

PROCEDURE TStack.Dec(VAR i : integer);

BEGIN
  i := i - 1;
END;


PROCEDURE TRealStack.Push(Item : real);

VAR i : integer;

BEGIN
   Inc(height);
   FOR i := 4 DOWNTO 2 DO
      Regs[i] := Regs[i-1];

   Regs[1] := Item;
END; { RealStack.Push }
FUNCTION TRealStack.Pop : real;

VAR i : integer;
BEGIN
   IF height > 0 THEN BEGIN
```

```
      Pop := Regs[1];
      FOR i := 1 TO 3 DO

         Regs[i] := Regs[i+1];
      Dec(height);
      ErrorFlag := FALSE;
   END
   ELSE BEGIN
      Pop := 1.0E+30;
      ErrorFlag := TRUE
   END;

END; { TRealStack.Pop }
PROCEDURE TRealStack.Add;

VAR i : integer;

BEGIN
  Regs[1] := Regs[1] + Regs[2];
  FOR i := 2 TO 3 DO
    Regs[i] := Regs[i+1];
END; { TRealStack.Add }

PROCEDURE THPStack.Add;

VAR i : integer;

BEGIN
  LastX := Regs[1];
  Regs[1] := Regs[1] + Regs[2];
  FOR i := 2 TO 3 DO
    Regs[i] := Regs[i+1];
END;{ THPStack.Add }

PROCEDURE TIntStack.Push(Item : integer);

VAR i : integer;

BEGIN
   Inc(height);
   FOR i := 4 DOWNTO 2 DO
      Regs[i] := Regs[i-1];

   Regs[1] := Item;
END; { TIntStack.Push }


FUNCTION TIntStack.Pop : integer;

VAR i : integer;

BEGIN
   IF height > 0 THEN BEGIN
      Pop := Regs[1];
      FOR i := 1 TO 3 DO
         Regs[i] := Regs[i+1];
      Dec(height);
      ErrorFlag := FALSE
   END
   ELSE BEGIN
     Pop := -32767;
     ErrorFlag := TRUE
   END;
END; { TIntStack.Pop }

PROCEDURE TIntStack.Add;

VAR i : integer;

BEGIN
  Regs[1] := Regs[1] + Regs[2];
  FOR i := 2 TO 3 DO
    Regs[i] := Regs[i+1];
END; { TIntStack.Add }

{ ---------------- declarations of variables ------------------- }

VAR RealStack : TRealStack;
    IntStack : TIntStack;
    { HPStack : THPStack; }
    X : real;
    I : integer;
    ch : char;


BEGIN
  NEW(RealStack);
  NEW(IntStack);
  NEW(HPStack);
  { exercise real-type stack }
  RealStack.IStack;
  RealStack.Push(1.0);
  RealStack.Push(2.0);
  RealStack.Push(3.0);
  RealStack.Push(4.0);
  RealStack.Add;
  X := RealStack.Pop;
  WRITELN('X = ',X); WRITELN;
  { use HPStack }
  HPStack.IStack;
  HPStack.Push(0.0);
  HPStack.Push(0.0);
  HPStack.Push(3.0);
  HPStack.Push(4.0);
  INHERITED HPStack.Add;
  X := HPStack.Pop;
  WRITELN('X = ',X);
  WRITELN('LastX = ',HPStack.LastX);
  WRITELN;
  { exercise integer-type stack }
  IntStack.IStack;
  IntStack.Push(1);
  IntStack.Push(2);
  IntStack.Push(3);
  IntStack.Push(4);
  IntStack.Add;
  I := IntStack.Pop;
  WRITELN('I = ',I); WRITELN;
  readln(ch);
END.
```

A Preemptive Multitasking Kernel Continued, Lattice dBC, and Compiler Controversies



by Allen Holub

Last month I presented the users' guide and part of the code for a small, preemptive multitasking kernel. This month's column finishes up with the rest of the code and a description of how the subroutines work. The code appears in Listings One to Seven on pages 110–123 of the December 1987 issue.

I can only hope to give you a bare-bones introduction to operating system innards here. If you're starting out from scratch and want to pursue the matter further, I can recommend two excellent books. If you're interested in kernels only (which is often the case for ROM-based systems), Ted Biggerstaff's *System Software Tools* (Englewood Cliffs, N.J.: Prentice-Hall, 1986) presents both the underlying theory and the complete C source for a small multitasking kernel that runs on the IBM PC. Biggerstaff's OS includes a primitive screen I/O system, but hard stuff—such as the disk I/O—is relegated to DOS. Unlike my kernel, though, Biggerstaff's program lets you run other programs as subprocesses.

For a more in-depth introduction to operating systems in general, Andrew Tanenbaum's *Operating Systems: Design and Implementation* (Englewood Cliffs, N.J.: Prentice-Hall, 1987) is the best book on the subject that I've ever seen. It's both very complete in its coverage of the subject and extremely well written. (It's not often that a textbook is readable, but this book is a glowing exception to the obfuscation-is-better rule.) Over the course of the book, Tanenbaum develops a complete Unix look-alike system, called MINIX, that runs on an IBM PC/XT or AT. He presents all the underlying theory in considerable depth as well as a complete implementation (in C) of the MINIX kernel. He covers virtually every aspect of operating system design, from the lowest-level disk driver (which interfaces directly to the hardware) up to the context-swapping code.

The book includes the full sources to the kernel, and if you spring for the accompanying disk (it's $80 from Prentice-Hall), you get an executable operating system and full sources for it and most of the other programs you need to actually use the operating system. (The disk contains a C compiler and an assembler, but you don't get the sources for these). The system provides something like 65 commands—the basic stuff such as cp, chmod, and so on and big stuff as well, such as a shell, an editor, grep, tar, uniq, roff, sort, pr, make, and ar. All these run under MINIX, of course, not DOS. But for $80 this is one of the deals of the century. Unlike Gnu, a public-domain Unix look-alike system, MINIX is not vaporware. In fact, the $80 you pay for MINIX is less than what you pay as a media fee when you get the "free" copy of Gnu.

Operating System Organization

As I mentioned last month, you can look at an interrupt-driven I/O system as an operating system. That is, each interrupt-service routine is a task, and the context swapping (changing from one task to another) is performed by the hardware every time an interrupt comes along. The task's priority is hard-wired into the hardware.

In most operating systems, however, context swaps are done in other ways: either a running task voluntarily yields (gives up control) to another task or a timer interrupt comes along and the running task is preempted (control is involuntarily taken from it). The part of the operating system that takes care of the swapping (and of figuring out which of several tasks should get control at any given moment) is called the kernel.

There are several ways to organize an operating system, two of which are shown in Figure 1, page 74. In the top diagram, the system is organized in onion-skin-like layers. Unix and MS-DOS are both examples of this organization. All I/O is done through the kernel, and access to the interrupts is done through the I/O system. The advantage of this approach is that it's easy to manage system resources—for example, you don't have to worry about two tasks both trying to write to the disk simultaneously; the kernel will act as a traffic cop.

The second diagram in Figure 1 shows how my own system is organized. This organization is mandated by the fact that I'm using DOS to do my I/O and that the kernel is part of the running program, not part of DOS itself. Resource management is a real problem in my system because DOS is not itself reentrant. It's up to the running task to assure that it cannot be preempted while it's performing a DOS call. You can do this in one of three ways.

The easiest method is to block all other tasks (the current task retains control of the CPU) using the *t_block( )* and *t_release( )* system calls. The problem here is that an I/O intensive task may use up an inordinate amount of system time.

An alternate approach is to use an exclusion semaphore. A semaphore is a length 1 message queue, created with a *t_makequeue( )*

(continued from page 72)

system call. Normally a message is waiting at the queue. When a task wants a resource, it pends (waits for a message to arrive) at this queue by calling *t_wait( )*. When the task has finished with the resource, it posts the message back to the queue using *t_send( )*.

A third approach puts another level of isolation into the picture. Here, a special task is in charge of every resource. A second task sends a request to the resource task (again using the message mechanism), which performs an operation and then sends the result back to the calling task. For example, a single task might be in charge of all screen I/O. When a task wanted to send something to the screen, it would just post a message at the screen task's input queue (the message would probably be a pointer to a string) and the screen task would do the actual I/O. Another example is a disk I/O task. Here the requesting task would send a read-request message to the disk manager. This message would contain some sort of file descriptor, a count of the number of bytes to read, a pointer to a buffer, and the address of a queue to which the disk manager should send a response. The manager would fill the buffer and return a "done" message to the indicated return address when the disk access was complete.

The resource-manager approach is often the best for two reasons. First, messages can't possibly get intermingled because the screen task will process them in the order received. Second, the I/O process can be done in parallel with other tasks. For example, the disk manager could start up a DMA transfer and then suspend itself by waiting on a queue. The interrupt-service routine that's triggered by the DMA controller when the transfer is done posts a message to this same queue. In the interim, however, the disk manager is suspended, which means that other tasks can be active. Had you used the block/release strategy or an exclusion semaphore, all activity would stop while the active task waited for the transfer to complete. The kernel itself contains no resource managers, however. You'll have to write your own.

The Data Structures

So, what exactly is a task? Individual tasks (or processes) consist of several parts. First is the code itself. For reasons that will be apparent shortly, this code must be reentrant. That is, you must write it as if it were a recursive function—at least in terms of static-variable usage and so forth. If you're careful, several tasks can share the same code (because, like recursive subroutines, they all have unique stacks—more on this momentarily). The second part of a task is a task control block, or TCB. The TCB that's used by the kernel is defined in kernel.h (Listing One) starting on line 60. It's also shown in Example 1, page 78.

The TCB is shown graphically in Figure 2, below. The TCB structure forms a header, and the remainder of the structure is memory that will be used for a task's stack. The size of the stack is determined at task-create time (it's passed into *t_create( )* as an argument). TCBs (or pointers to them) are stored in one of three places: in an active list, which is a priority queue consisting of all tasks that have been preempted and are waiting to be activated; queued up at a message (I'll look at this process in a moment); or, if the task is the currently running task, a pointer to the task is held in the global variable *T_active*.

Several things happen when a timer interrupt comes along. First, the various message queues are examined, and if any of the tasks waiting at these queues have timed-out, they are added to the active list. Next, the priority of the running task is compared with the highest-priority task in the active list, and if the latter is higher, a context swap is

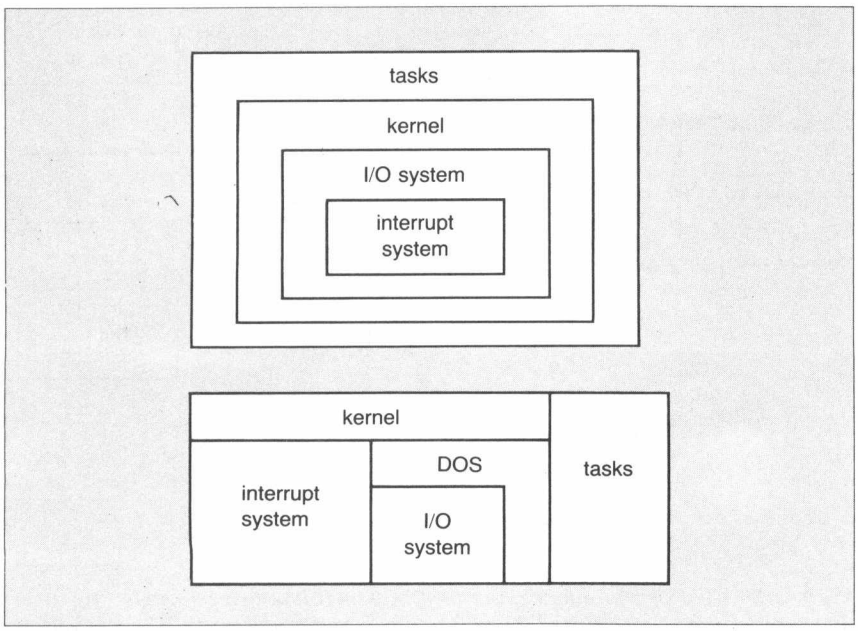


Figure 1: *Two ways in which to organize an operating system*

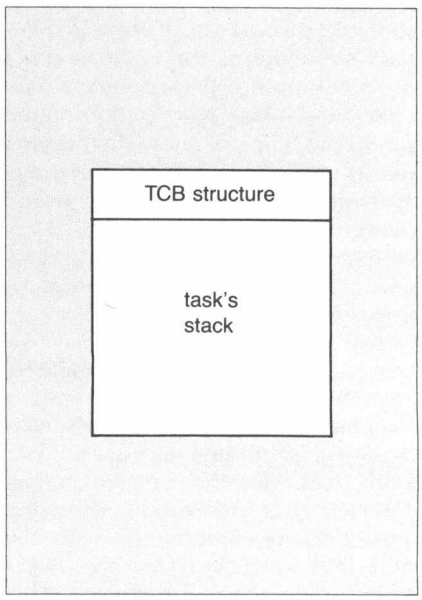


Figure 2: *Graphical representation of the TCB used by the kernel*

C CHEST

(continued from page 74)

performed as follows. The value of the system clock (held in *T_clock*) is copied into the *timestamp* field of the structure. Then all the registers except *SS* and *SP* are pushed onto the current task's stack. Next, the *SS* and *SP* registers are copied into the *ss* and *sp* fields of the TCB. Now the new task is dequeued from the active list and the old task is enqueued in its place. The new task is installed, first by copying the *ss* and *sp* fields into the corresponding registers (thereby making the new task's stack the active stack) and then by popping all the registers. If you've done a context swap, the popped registers will be those that were saved the last time the task was suspended, not the registers that were saved when you entered the timer interrupt-service routine (which are in the suspended task's TCB).

Note that the time of last preemption is considered when a task's priority is examined. That is, if two tasks have the same priority, then the one that has been waiting longest is considered the higher-priority task. This is how round-robin scheduling works.

The other fields in the TCB are used for debugging and message passing. The three debugging fields are *tag*, which points to the string that you passed into *t_create( )*; *initial_sp*, which is the initial value of the stack pointer; and *status*, which tells you if a task is currently waiting for a message or not. The kernel doesn't use these fields for anything. The *tag* field is particularly useful because it lets you see which task is actually attached to a given TCB—you can pass the task name to *tcreate( )*.

The remaining fields are used for the message-passing system. The *next* field is used to maintain a queue of tasks, all of which are waiting at the same message queue. The TCBs of these tasks are arranged as a linked list, with new tasks being added to the end of the list. The first TCB in the list gets a message when one arrives at the queue. This is accomplished by unlinking it from the list, setting the *msg* field to point

```
typedef struct tcb
{
    void          **sp;          /* Task-swap stuff */
    unsigned      ss;

    unsigned      priority;
    unsigned long timestamp;

    unsigned      wait;          /* Message-management
stuff */
    struct tcb    *next;
    void          *msg;

    int           status;        /* debugging stuff */
    char          *tag;
    void          **initial_sp

    void          *stack[1];     /* Base of system stack
*/
}
TCB;
```

Example 1: *The TCB used by the kernel*

```
typedef struct t_queue
{
    int             signature;
    struct t_queue *next;

    TCB             *task_h;     /* Task queue */
    TCB             *task_t;

    int             q_size;      /* Message queue */
    int             numele;
    void            **headp;
    void            **tailp;
    void            *queue[1];
}
T_QUEUE;
```

Example 2: *The* T_QUEUE *structure*

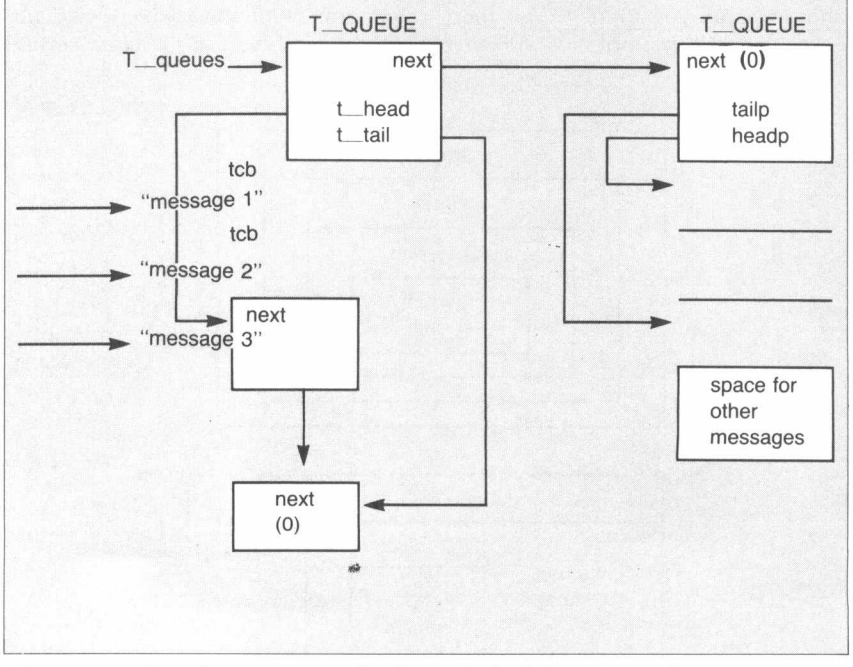


Figure 3: *A list of two queues, the first of which has two tasks waiting at it and the second of which has three messages enqueued and no tasks waiting. Only relevant fields are shown.*

(continued from page 79)

at the message, inserting the task into the active list, and then forcing a reschedule (as if a timer interrupt had come along). The *wait* field is a counting semaphore. It is initialized to the time-out value that you pass to *t_send( )* and is decremented on every timer interrupt. When it gets to 0, the task is removed from the message queue and put back into the active list with the *msg* field set to *NULL*.

The next data structure of interest is the *T_QUEUE* structure, defined on lines 98–113 of Listing One and in Example 2, page 79. *T_QUEUE* is a variable-length structure similar to a TCB. Here, however, the area at the end of the queue is the message queue itself (rather than the stack). *Queue* is the first cell of that area. The *headp* and *tailp* fields point at the head and tail of the message queue and *numele* is the number of messages currently in the queue. The *q_size* field is the maximum number of messages that the queue can hold. Waiting tasks are queued up using the *task_h* and *task_t* fields. The former points to the first element in the linked list and the latter points at the last element.

Self-modifying code turned out to be an easy work-around

The queues themselves are also arranged in a linked list, put together in the order in which the queues are created. The timer interrupt-service routine chases down this second list looking for timed-out tasks. The *next* field points to the next queue in that list; the situation is illustrated in Figure 3, page 79. Finally, the *signature* field contains an arbitrary number that's used by some of the queue functions to make sure that the *T_QUEUE* pointer passed to them is valid.

The Subroutines

Once you understand the data structures, the actual code is pretty straightforward. Schedule.asm (Listing Two) contains the timer-interrupt-related stuff. It is very similar to the *speedup( )* subroutine described in the September 1987 C Chest, so I won't go into details here.

_T_speedup( ) (line 182) speeds up the IBM PC system clock by an indicated factor and installs an interrupt-service routine that does context swaps when required; *_t_slowdown( )* (line 241) slows the clock down again and removes the scheduler. *T_block( )* and *t_release( )* (lines 165–175) just set or clear a global flag (*blocked*, declared on line

118) that will be examined by the interrupt-service routine, *serv*, on lines 281–345. The flag is checked on line 285, and if it's set, *numblk* is incremented (for statistics purposes only—it tells you how many interrupts were blocked) and control passes to *servexit* on line 327. The code on lines 328–240 vectors to the default DOS interrupt-service routine every *N* interrupts, where *N* is the original speedup factor; otherwise, a nonspecific EOI is issued to the timer chip and an *IRET* is executed.

The actual context swap is straightforward. Registers are pushed on lines 295–302. Note that the *CS*, *IP*, and flag registers are saved by the hardware as part of the interrupt-service procedure, so you don't have to push them again here. The current *SP* and *SS* are saved on lines 306 and 307, a local stack is then installed, and the C subroutine *_t_reschedule( )* is called on line 313. This subroutine (starting on line 512 of Listing Six) scans the list of messages and times out appropriate tasks (in the *for* loop on lines 532–550). It enqueues the running task's TCB and sets *T_active* to the new task's TCB when a context swap is required (on line 567).

Now control returns to the assembly-language code. The new TCB (which is the same as the old one if no swap is necessary) is fetched on line 315 of Listing One. Then the stack is restored, registers are popped, and you keep going. So, if you're doing a context swap, you'll save registers on the old TCB and then restore them from the new one. Note that the *CS*, *IP*, and flag registers are restored as part of the *IRET* instruction.

All the other assembly-language stuff is in swap.asm (Listing Three). The routines in this file are not particularly well structured—that is they save space by sharing code. One subroutine will jump into the middle of another. The central routine is *_t_swap_in( )* on lines 172–242. It is passed a pointer to a new TCB. It suspends the current task and installs a new one as if it were the timer interrupt-service routine. Again, registers are saved (lines 183–198), *T_active* is modified (line 216), and the new task is installed by popping registers (lines 227–242). Note that the subroutine's return address is used as the old task's saved *IP* register. This means that when control is restored to the task, you'll be back in the calling subroutine, just after the *_t_swap_in( )* call.

The other swap-related functions are *_t_install( )* (lines 142–168), which deletes the current task and installs a new one, and *_t_ shazam( )*, which starts up multitasking. Both of these jump into the middle of *_t_swap_in* (to the *shazam* label on line 223) to install the new task. *_T_shazam* also changes the stack probe subroutine, *chkstk*, which is called at the start of all normal subroutines. I couldn't use the normal *chkstk* because the task stacks are in strange places (in the middle of the heap in fact), so the default *chkstk* would always fail. The new routine checks the current

(continued from page 82)

SP against the current TCB's stack base and reports an error if the stack gets too large.

Note that the installation procedure involves actually modifying the first couple of instructions of the default *chkstk* subroutine to jump to the new one (on lines 271–274). I know this is a kludge, but Version 4.0 of the Microsoft compiler (and early betas of Version 5.0) made it virtually impossible to link in my own *chkstk*. (At least, I tried for a couple of hours with no success.) Self-modifying code turned out to be an easy work-around, but it's a stopgap measure, and I mean to fix it once I figure out how.

The final routine of interest is *__t_stop( )* (lines 94–138), which is called to terminate multitasking. It causes control to pass back to the instruction following the original *t_start* call—the return address was saved by *t_start( )*. *T_stop( )*'s argument is passed back as the return value of *t_start( )*. That is, a call to *t_stop(5)* will cause *t_start( )* to return a value of 5 to its caller. The situation is analogous to an *exit( )* call.

The remainder of the code is straightforward enough and sufficiently commented that there's no need to go through it here. I do suggest that you read it, though. It's pretty interesting. I'll finish with a caveat. I've used the routines printed here in a couple of application programs, and they seem to work fine. These program all use my own direct video I/O functions to write to the screen, however, and they don't do any disk I/O. It's possible to use DOS, but you'll have to implement one of the resource management strategies discussed earlier (or use a nonpreemptive system). Do not call DOS directly from a task without assuring that the task has control of the system for the life of the DOS call.

DBASE has spread like a disease through the small-business community.

I'm reasonably sure that the code presented here is bug free, but I won't swear to it. If you find a bug please report it to me c/o Software Engineering, P.O. Box 5679, Berkeley, CA 94705. Use E-mail to leave a message on CompuServe (because I don't log on reliably enough to guarantee that the message won't roll off the forum message board before I see it). Post a message on the forum, though, so everybody can see it.

Priority-Queue Routines in the Kernel

In addition to the various subroutines presented this and last month, the kernel uses the priority-queue routines from the June 1987 C Chest—you should look at that article for detailed information on how these routines work. I've modified the original code just a bit, however.

The *pq_look( )* routine now returns *NULL* if the queue is empty. It used to return garbage in this situation. I've also added a "replace" routine that replaces the highest-priority element in the queue with a new element, reheaps, and then returns the former top element. It's as if you did a delete of the *old* element followed by an insert of the new one, but it takes less time than would two operations. Both modifications are shown in Example 3. Also note that a bug fix that affects *pq_insert( )* was mentioned in the August 1987 C Chest, page 108.

```
void    *pq_look( queue )
PQ      *queue;
{
    return queue->nitems ?  queue->heap : NULL ;
}

/*-------------------------------------------*/

int  pq_replace( p, target, item )
PQ   *p;
void *item, *target;
{
    int slots_in_use;

    if( slots_in_use = p->nitems )
    {
        memcpy( target,  p->heap, p->itemsize );
        memcpy( p->heap, item,    p->itemsize );
        reheap_down( p, p->heap );
    }

    return slots_in_use ;
}
```

Example 3: *The routines* pq_look( ) *and* pq_replace( )

Nifty Stuff: The Lattice dBC Library

Every programmer should have a database management library in his or her toolbox. This month I'll look at one such library—I'll look at others in future columns.

There are a host of database packages on the market, all with their own set of strengths and weaknesses. One of the weakest, from a database management point of view, is Ashton Tate's dBASE III. Unfortunately dBASE has spread like a disease throughout the small-business community. Consequently, it's often necessary to put together a dBASE-compatible application program, even if you'd rather use one of the more efficient database packages.

DBASE III is actually a programming language of sorts. In practice, it's so limited and hard to use that

no sane person would write a complicated application in it (at least not if they want to retain their sanity). As a consequence, several fourth-generation languages that support the dBASE file structure, such as FoxBASE and Quicksilver, have sprung up. These products have problems, too—they're limited in one way or another. I want to write my applications in C, not in some inherently inefficient and much too limited 4GL that's really designed for nonprogrammers. My programs will be faster and I can use all the other C functions (such as window management packages) that are at my disposal.

The Lattice dBC III and dBC III Plus packages provide a solution to this problem. These packages are subroutine libraries that you can link into your programs to access dBASE III or dBASE III PLUS databases. The version that I used links to Microsoft-generated code, not Lattice-generated (though versions for the Lattice compiler are available, of course). Versions of the package are available for versions of dBASE down to dBASE II. The main difference between the III and the III Plus version of dBC is networking support (you can lock whole databases, individual records, and bytes within the record). The package comes with small-, medium-, and large-model libraries as well as a demo program (and the sources for the demo program). Though the version that I used expected Microsoft C, Version 4.0, it linked into a program compiled under Version 5.0 without difficulty. (You can't use the new combined libraries, however—the components have to be on the disk.)

I have very mixed feelings about this package. On the plus side, it really does what's required. It took me an hour or so to read the manual, and a few hours later I had a working C program that was accessing a database created by dBASE III PLUS. You should have no difficulty doing the same, even if you've never worked with dBASE itself. Keep all this in mind as you read the negative comments that follow. Though the package has problems, most of them are fixable and dBC is really useful.

Unfortunately, this ease of use has more to do with the simplistic structure of a dBASE III file than with the Lattice documentation. A chapter that described the organization of dBASE .dbf and .ndx files would have been very helpful. Data is stored in strange ways and an in-depth description of these methods would also be useful. (There's a description of dBASE .dbf and .ndx files in Jeff Walden's *File Formats for Popular PC Software* [New York: Wiley, 1986.])

Some of this information can be gleaned from the subroutine descriptions, but I'd like it all in one place. The manual is poorly organized and hard to use as a reference. For example, dBC functions naturally break into four categories: functions to manipulate the database itself, functions to manipulate the index files, functions to manipulate the memo files, and various formatting functions. The manual, however, is organized alphabetically (generally my preference), but there is no functional index. As a consequence it's hard to find the function you need if all you know is what it does. The subroutine naming conventions are so strange that it's hard to guess what the name will be. The problems with the documentation are compounded by occasional errors in the examples. (One of them has a couple of stars where none belong, an error that caused me about ten minutes of unnecessary head scratching.)

The subroutines themselves are sometimes rather low level. The low-level functions should be included in the package, but higher-level ones are really required, too. For example, there's a host of functions that convert strings and numbers to and from the formats required in dBASE files. Fine, but as a C programmer, I want *sprintf( )*- and *sscanf( )*-like subroutines that would take care of all the details of the conversion for me. Ideally, you'd pass these functions a database pointer, a format string that identified the fields you wanted to access, and several objects to convert—something like this:

```
dprintf( db_file,
        "%name %street %zip",
        "J. P. Morgan", "Easy St.", 09311 );
```

and the function would do the rest for you. No such function is provided, however.

The dBC programming interface is very amateurish and harder to use than is necessary. I'd expect this sort of thing from users' group software but not from professional programmers. There are several problems. First, Lattice has implemented functions with C-like names but with nonstandard interfaces. For example, the function used to open a database is obviously modeled after *open( )*. It's called with:

The package does the job and is easy enough to use.

```
int dBopen( filename, mode, dbffd )

char *filename;
int mode;
char **dbffd;
```

There are several problems here. The main one is that you get back the file descriptor by passing in a pointer to a place to put it (*dbffd*). *DBopen( )* returns an error code, but that's it. I also don't like the declaration of a file descriptor as character pointer. It obviously isn't a string and shouldn't be declared as such. I'd prefer either a *void* pointer or a special type (such as *FILE*).

Were I to rewrite the interface to this function, it would look like this:

```
DBFILE *dBopen( filename, mode )

char *filename;
int mode;
```

It would return –1 on error and a file descriptor on success; there

would be a *db_errno* that would hold an error code and a *db_perror( )* that would print an error message. That is, the dBC function should be identical to *open( )*, except that it should open a database rather than a normal file. If you're going to use this package a lot, it would be worthwhile to write a glue library that mapped the weird calling conventions to something more reasonable.

The second problem is the function names themselves. The names are the too-cryptic abbreviations popular with inexperienced C programmers (such as *_dbcmsiz( )* or *dBdbfbuf( )*). There seems to be a rationale to the naming conventions, but that rationale is never made clear.

Finally, the interface has several built-in inefficiencies. For example, many values are returned from subroutines in a *char* rather than an *int*. (Usually you pass in a pointer to a *char*, which is filled by the subroutine.) This is a classic mistake made by novice C programmers who think they're saving space by shoving data into *char*s. They're wrong. The compiler must convert all *char*s to *int*s before they can be used in an expression. This conversion is always performed, even in expressions that use nothing but *char*-size variables. Not only does the type conversion take time but also the extra code needed for the conversion is usually far larger than the data space saved. The only valid use for C's *char* type is an array of the things.

Frankly, the poor-quality interface makes me wonder about the quality of the rest of the code. The package does work and has been reliable in all the applications that I've written, but I don't really trust it.

In spite of the foregoing, though, I recommend this product to those of you who need to write Ashton-Tate-compatible programs. I intend to keep using it myself in these situations. The package does the job and is easy enough to use. It lets you put together a working program in a reasonable amount of time with a minimum of fuss. Moreover, you can do it all in C, without having to learn yet another programming language. To make dBC really usable, however, you'll need to add a layer of glue functions to make the program interface more reasonable.

I wouldn't use dBC III Plus if I didn't need dBASE compatibility, however. The ISAM model used by Ashton-Tate is not great for most applications. The databases themselves are poorly structured, generally hard to use, and slow. (The speed is Ashton-Tate's problem, not Lattice's. Any program made with Lattice's package will be an order of magnitude faster than the equivalent dBASE III program. Nonetheless, a database program that used a different model altogether would probably be faster still.) Moreover, the Lattice package is marred by the poor quality of both the documentation and the programming interface. Though you could use dBC for any general-purpose database program, I wouldn't.

Availability

All source code for the multitasking kernel described both this and last month (including the priority-queue stuff) is available for $30 on an IBM PC 5¼" disk from Software Engineering Consultants, P.O. Box 5679, Berkeley, CA 94705. Include local sales tax if you're ordering from California.

In addition to the kernel code and the priority-queue routines, the disk includes an enhanced version of the curses window I/O package described in the July 1987 C Chest. Because the enhanced curses uses direct video reads and writes rather than going through DOS, it's useful in multitasking applications that can't use the DOS I/O functions. This version of curses supports overlapping windows (though you can only write to the top one) and lets you delete and move windows. In addition, it lets you create boxed windows (Unix's curses doesn't).

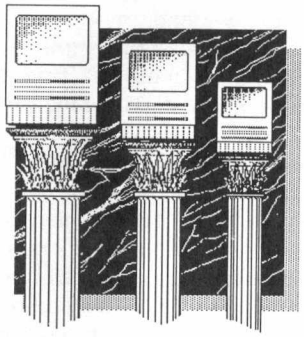
The (unmodified) priority-queue routines are also available on CompuServe in DL1 of the DDJ FORUM. The file is called QUE2.C.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

TO THE MACS

Introduction, Mac Sources, Lightspeed C, and Code Corner



by Stan Krute

It's been four years now since the Macintosh popped into view, heralded by that great Ridley Scott Superbowl commercial. Though there was obvious brilliance to the design, there were also strong whiffs of arrogance and hype. But, hey, I've been accused of the latter myself. There was nothing to do but pop out of the hills and take a closer look.

I cruised over the Siskiyous to see my Apple dealer buddy, John Manzer. I got to the store, chewed the fat a few minutes, scanned the marketing propaganda, nosed the technical specs, then plunked down at the machine. Cynical musings twisted my mind, but what the hell, let's start 'er up.

I didn't like the locked hardware. I didn't like the one drive. I didn't like the lack of a hard disk. I didn't like *the price*. I didn't like the (hah hah) wide selection of printers. I didn't like the yuppistic overtones.

But I loved the machine. It was fun to use. Oh, there were flaws, but they seemed minor compared to what the creators got right. Above all, the interface snapped. It didn't have a speed snap—not yet—but rather the feel-of-a-fine-tool kind of snap. The Macintosh communicates via clean visual metaphors, and that's a channel with a lot of bandwidth. Something about the Macintosh interface just feels good, like soft light filtering through a redwood forest or playful kittens careening and bouncing about the world. People needed to use computers like this. I needed to program computers like this. Yow!

Four years of good nurturing has led to some lovely growth. We've got a wide array of languages, detailed system documentation, the LaserWriter, Mac IIs, Hypercard, MultiFinder, and some remarkable application software. The platform's been consolidated, and the best is yet to come. Happy birthday, Mac! Blow out those candles, eat up that cake, chug down the Jolt, get crazy with your buddies. Hell, Maddie Hayes's got you in her office: Y'all done good.

The Doc Gets a New Column

And so *Dr. Dobb's* gets a Mac column. What'll I do here?

1. Take note of well-done applications and extensions to the user interface.
2. Review a wide array of Mac programming tools: software and works on paper.
3. Talk with, and about the work of, innovative Mac programmers.
4. Discuss some of the more interesting algorithms and data structures contained in the Mac ROM/OS. This thing's a graduate course in programming, with interesting tidbits lurking between every *LINK/UNLK* pair.
5. Write some code. Mac programming's the most addictive fun I've had in the innards of a machine. The universe of the Mac ROM/OS is quite dynamic, so there's a premium—nay, an imperative—on programming that's clean, concise, and careful. Yield to that imperative, then combine it with an interface design that syncs with the Mac paradigm, and you get applications that not only work but that are also fun, easy to use, empower your users, and smack of elegance.
6. Provide access details. I'll always give you a box (see page 106, for example) filled with information that'll help you get hold of items mentioned in that month's column.

Reviews, Criticism, Objectivity

It's a lot of hard work to get a book or software product on the market. I feel a special obligation to creators to be scrupulously fair with any review/comments/criticism of a work. Print's powerful stuff. If I think something is seriously flawed, I won't even bother to mention it here; I prefer to send a quiet note detailing my qualms directly to the publisher. I'd rather put this column's energy into feeding awareness of the good stuff.

A note on objectivity: I'm lucky enough to know and/or have worked with some of the people whose products I may mention. But it does nobody any good if I let that shade my opinions. On the other hand, I don't want to ignore a good product just because I've had something to do with it. So I'll always mention any close connections I've got to a particular item in an objectivity note. Just know that it's done to help you weigh my opinions, not as name dropping.

Getting Up To Speed

The code samples I'll be showing aren't for raw beginners. This is *DDJ*, after all. But it's easier to get up to Mac programming speed now than it was in the early days. A lot of resources are available to help you cruise the learning curve. Here's a minimal list:

1. Join APDA, the Apple Programmer's and Developer's Association. Godchild of Dan Cochran and Dave Lingwood, this is a one-stop source for draft and finished copies of Apple documentation and development tools as well as a wide variety of third-party products. Dues are a reasonable $20 per year, it has an

TO THE MACS
(continued from page 90)

800 phone number, and you can charge to plastic.

2. If you're developing commercial products, try to become a certified Apple developer. Most important, this gives you access to Apple's electronic-mail technical support. Within the corporate constraints, the remarkable tech support humans will help you work through most any problem. Answers come within 24 hours. Other certified developer pluses: marketing assistance, developers' conferences, discounts on development hardware, and a tinge of credibility.

3. Get *Inside Macintosh* and its descendants. If the Pulitzers had a technical writing category, *Inside Mac* would own a prize. Caroline Rose and her cohorts and descendants have given us the most comprehensive insight into a complex cybernetic system yet seen. This is the starting point for all Macintosh programming. Take a look at the APDA newsletter for the latest volume count. The only flaw is a lack of practical examples, but other folks have filled that gap (see next item).

4. Add at least the following five books to your library: Scott Knaster's *How to Write Macintosh Software*, Dan Weston's *The Complete Book of Macintosh Assembly Language Programming (Volumes I and II)*, *The Best of MacTutor*, and *The Complete MacTutor*. Other fine Mac programming books are available, but these five are classics. They give you the practical examples that *Inside Macintosh* lacks. And, if you share my lack of photographic memory, you'll also want some language references. I like the handy little *Signetics S68000 User's Guide* for 68000 assembly language and Harbison and Steele's *C: A Reference Manual*. (Objectivity note: Dan Weston is a longtime friend and fellow traveler.)

5. After you've reupped with *DDJ*, subscribe to *MacTutor*. It's one great Macintosh programming magazine, filled each month with nerdly little programming goodies.

6. Put together an array of development tools. Plenty of good ones are available, and every now and then, I'll review some here in the column. After all, I'm a language junkie; one of the perks of this gig is feeding the addiction guilt-free.

Choosing development tools is pretty personal. With one exception (now justly dead in the market), I haven't hit a Mac programming tool that someone wouldn't find useful in some context. The Mac environment must have some kind of inspirational effect. You'll have to follow the usual path to find personally amenable tools: talk to friends, read reviews, scan the ads, ask questions on the networks, play around. For what it's worth, here's a commented list of what I currently find myself using; note that most of my Mac work is done in C, with 68000 assembly language for speed tweaks and writing specialized code resources.

- C compiler: Lightspeed C 2.11—blazingly fast, holds to standards, feels good.
- 68000 assembler: MDS 2.1—I started here and have found no reason to move on; now marketed as the Consulair 68000 Development System.
- Debugger: TMON 2.8—clean, simple, powerful, can survive a lot of weirdness.
- Text editor: QUED/M 2.04—solid, feature-packed, useful macro language, very nice.
- Resource editor: ResEdit 1.1B1—one of the unsung great hacks, this Apple-produced program is the Mac-like way to create and manage resources.
- Disk and file editor: Fedit Plus—does anything you can think of to disks and files, fast and accurate with a very clean interface. (Objectivity note: I worked on the latest Fedit Plus documentation.)
- Code snooper: MacNosy—allows intelligent examination of any piece of code you can specify, including and especially the ROM. (Objectivity note: I worked on the [little yellow book] MacNosy documentation.)

7. Get a Mac with a hard drive and as much memory as you can afford. Anything less will drive you nutso fast. Hey, I oughtta know: I did my first Mac programming in assembly language on a 128K one-drive machine with 8-minute turnarounds. With a language such as Lightspeed C or Turbo Pascal on a multimeg SCSI machine, turnarounds drop down into the sub-30-second range.

Useful Mail-Order Sources

When I'm not traipsing around civilization as a cybernetic nomad, I live in the middle of nowhere, so I have to rely on mail-order sources to get programming books, software, and miscellaneous supplies. I've found a couple of good ones I'm happy to share with you.

For books, I use Computer Literacy. This bookstore carries just about everything, takes credit cards, and ships UPS the day you order. For software and supplies, I use Computerware. It specializes in the Mac, also takes credit cards and ships UPS quickly, and has an 800 phone number. Both these places have retail outlets well worth a visit if you're in Silicon Valley.

Don't Trash Your Old Mac

Apple's Macintosh upgrade path has been a little bumpy. A lot of folks still have 512s, possibly upgraded from 128s, and wonder whether it's worthwhile doing any further upgrading. Here's what I did: got Apple's 800K drive/128K ROM upgrade ($300 at an Apple dealer), then added SuperMac's Enhance board

(continued from page 93)

($500 installed at Fry's Electronics). Enhance brings your Mac up to 2 megabytes (expandable to 6.5 with high-capacity SIMMs), gives you a slight speed increase, and adds a SCSI port and a small internal fan.

I had one problem with the parasitic clip that plugs Enhance into the motherboard's 68000, but SuperMac kept Federal Expressing me replacements until we had the problem licked—no problems since then. I get a big grin on my face when my original 128K Mac comes up on a speedy hard disk with megabytes of RAM at its disposal.

Lightspeed C

It's hard to hold in my feelings on this product. Let's just say this: I love it. Michael Kahl and the rest of the Think Technologies crew have given us something wonderful. This thing is fast—makes me want to stick some flame decals on the Mac. Working in the LSC environment's a tasty treat, and I for one refuse to go back to anything slower or less capable.

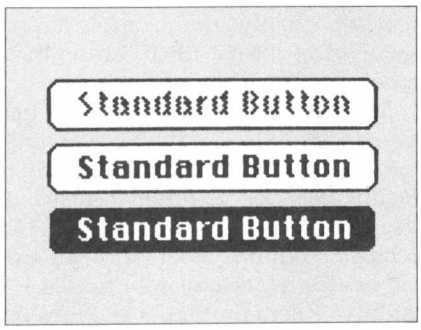


Figure 1: *Standard button in its three highlighted states—inactive, active, and highlighted*

Producing robust Mac code is (for me, at least) a highly iterative process. It bears repeating: Mac software exists in a very dynamic universe. The slightest coding miscue quickly propagates into screen-twisting madness. Debugging is tricky—well worth avoiding—so I like to write my code in snippets, testing and debugging each piece thoroughly before moving ahead. Lightspeed C, with its blazing turnaround speed, lets me do this painlessly.

A lot of attention's been paid to the product's details. Work goes on in a well-integrated project environment. Nitty little maintenance details are automated. The language, libraries, and header files hold closely to the relevant standards (Unix, Kernighan & Ritchie, Harbison & Steele, the evolving ANSI C, and *Inside Macintosh*). The editor's good—not quite so feature-laden as QUED/M but good—with powerful grep capabilities. The compiler puts out code that's fast and compact. It's easy to produce the various sorts of Macintosh code, with global and static variables available in each: double-clickable applications, desk accessories, device drivers, and code resources. In-line assembly-language code's allowed, with full access to the C name spaces. Resource file management is completely automated. Register variables are maximal: five data registers and three address registers. HFS and MultiFinder are well supported.

I recently had the pleasure of spending a September afternoon at Think headquarters doing free-form nerd talk with Michael Kahl (the prime Lightspeed C programmer), Andrew Singer (head of Think and coconceiver of Lightspeed), and Doreen Duplin (marketing/communications whiz). These are nice people in whom the joy of the great hack runs deep. Interesting backgrounds: Michael was a philosophy grad student before succumbing to the lure of machine logic. Andrew's known to many of us for his classic (and, sadly, out of print) Sherlock Holmes pastiche programming books *Elementary BASIC* and *Elementary Pascal*.

Recent releases of both Lightspeed C and Lightspeed Pascal (2.13 and 1.11A, respectively, as this column is written) have been maintenance releases, keeping the languages current with the latest Mac machinery and system software. In the works, though, are major new releases of both languages. Look for greater speed and, for Lightspeed C, powerful debugging capabilities. "It's time for another dose of the spectacular," quoth the Singer.

I wish I had room to give you a complete transcript of the afternoon's conversations. The Thinkers said a lot of smart stuff. Maybe in a future column. Meanwhile, take this as a bottom line: if you program the Mac in C, check out Lightspeed.

variation	content	border	highlighting via
0	text	outlined	inversion
1	text	outlined	content change
2	text	shadowed	inversion
3	text	shadowed	content change
4	PICT	bare	inversion
5	PICT	bare	content change
6	PICT	outlined	inversion
7	PICT	outlined	content change
8	PICT	shadowed	inversion
9	PICT	shadowed	content change
10	ICON	bare	inversion
11	ICON	bare	content change
12	ICON	outlined	inversion
13	ICON	outlined	content change
14	ICON	shadowed	inversion
15	ICON	shadowed	content change

Figure 2: RectCDEF*'s 16 button variations*

Code Corner

All right, time to get down to a little code hacking. My first project involves writing and using a custom control definition. Because of space constraints, I'll describe the project in two phases, continuing the discussion in next month's column.

Macintosh applications are rife with controls: buttons, scroll bars, check boxes, radio buttons, et al.

TO THE MACS
(continued from page 94)

Definitions for standard controls are built into the Mac ROM/OS—there's the standard button, for example. Clicking a standard button with the mouse makes things happen. Standard buttons have three basic states, each with a corresponding visual metaphor: inactive, when the button won't respond to a mouse click; active, when the button will respond to a mouse click; and highlighted, when the button's in the midst of being clicked. Figure 1, page 94, shows a standard button in each of these three states.

But you also have the ability to define, via a CDEF code resource, your own buttons. The CDEF resource can then be incorporated into an application and can be called upon whenever the application wants to put a button on the screen. Custom CDEFs are not very difficult to write and can provide a lot of flexibility at low memory cost. The CDEF I'll be showing you in this column, for example, is less than 1,400 bytes long yet it provides 16 new types of buttons—that's less than 88 bytes per button variation. Such a deal.

Development Details

I wrote my CDEF, called *rectCDEF*, in assembly language using MDS 2.1. That's because I wanted high execution speed and small code size. I wrote a demo application in C that shows off the 16 button types using Lightspeed C 2.11. Resources for the application were put together with ResEdit 1.1B1. PICTures for particular buttons were drawn in SuperPaint 1.0p, then transferred into ResEdit via the Scrapbook.

I first got the demo application up and running, albeit with just one control, that being of button variation 0 (see later). Then I started work on the CDEF. As I worked on the CDEF, I used an Exec JOB file to assemble the code, link it, turn it into a resource, then merge that resource into the demo application for testing. I worked on one variation at a time, adding a control of that type to the demo application, then fixing the CDEF to cover that case.

Any particular CDEF can have up to 16 variations. (Actually, you can hack in a few thousand, but that technique's for another article.) I used all 16 in *rectCDEF*. The *rectCDEF* buttons live in a rectangular world. A particular button variation can contain text, a picture, or an icon. Text can be in any font/size/style combination the Mac's capable of. A button variation can have a simple outline, a shadowed outline, or (unless it's a text variation) no outline. A button variation can indicate highlighting via inversion or a change of content.

Figure 2, page 94, details the 16 *rectCDEF* button variations. Figure 3, page 98, shows examples of each variation, with pictures of the active and highlighted states.

An Overview

The demo program, imaginatively named custom controls demo, puts up a modal dialog containing examples of *rectCDEF* buttons, then responds to button clicks. Figure 4, page 100, is a screen snapshot of the program's modal dialog. Using a modal dialog simplified the program's event-handling logic; it's a nice technique for bench testing new routines.

Figure 5, page 100, shows the files

TO THE MACS
(continued from page 96)

involved in the demo program. Custom controls demo PROJ is an LSC project file that contains the C source code file; custom controls demo.c (see Listing One, page 54); and MacTraps, the LSC library that hooks code into the Mac ROM/OS (see Figure 6, page 100). Custom controls demo.h, in Listing Two, page 64, is a file of private definitions for custom controls demo.c. Custom controls demo PROJ.rsrc (available on CompuServe and the *DDJ* listings disk) is a collection of program resources, including *rectCDEF*, that gets bound into the final application. It was put together with ResEdit. Finally, custom controls demo

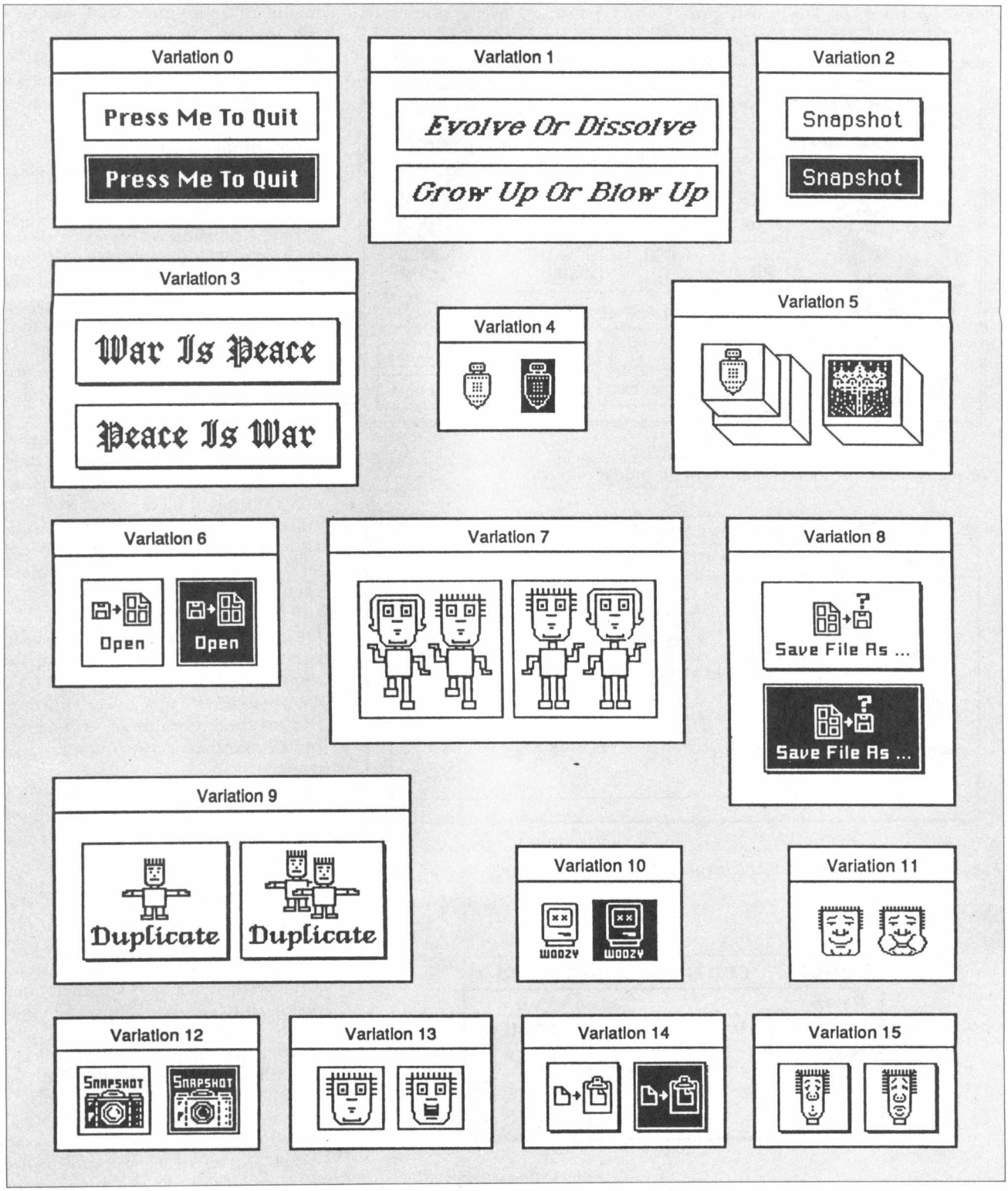


Figure 3: *Samples of the 16* rectCDEF *variations in active and highlighted states*

(continued from page 98)

is the double-clickable final application.

Although it's small and simple, custom controls demo.c follows the classic pattern of Macintosh programs. First come a few setup activities. Then the program sits in a main event loop, waiting for events of interest. When such an event occurs, the program figures out what's up, acts appropriately, then returns to the main event loop. At some point an event occurs that tells the program to pop out of the main event loop. Then come a few cleanup activities, followed by an exit to the OS shell.

Each control button in custom control demo's main modal dialog has a corresponding item number in the DITL resource that supplies the dialog. Figure 7, page 101, matches each item with its DITL item number. These numbers are given symbolic names in the header file custom control demo.h. The same numbers are used for the CNTL template resources that each DITL item points to.

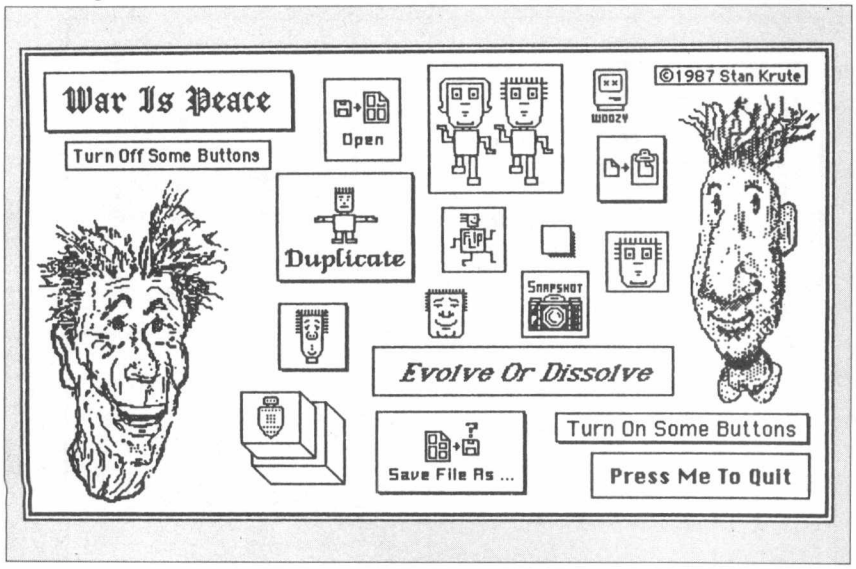


Figure 4: *Custom control demo's main modal dialog*

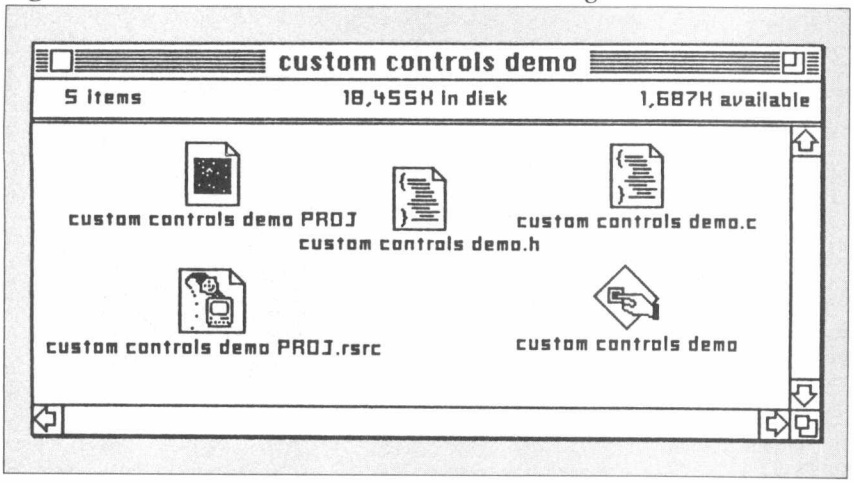


Figure 5: *Five files used by Lightspeed C to build the program custom controls demo*

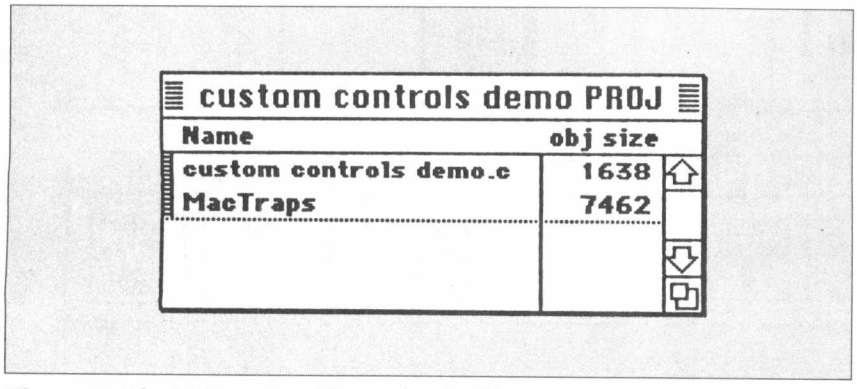


Figure 6: *The LSC project file used to build custom controls*

A Few Function Notes

If you've done your homework, the demo program should seem trivially simple, so I won't go into massive descriptive detail. That'll get saved for next month when it's time to cruise the *rectCDEF* assembly-language code. Here are a few notes:

main—Sets up the Mac managers, gets the modal dialog going, runs the main event loop, then cleans up and exits when all is done. Note the substitution of the ROM call *ModalDialog* for the usual *GetNextEvent* as the heart of the program's main event loop.

inititializeManagers—Grabs some master pointers, forces the heap to grow and clean itself, gets the ROM/OS managers up and running, flushes the event queue, and brings up the standard arrow cursor.

studyAndSetEnvironment—Figures out the size of the screen and menu bar. This information is used later on to position windows neatly.

getThatDialogCookin—Brings the main modal dialog into memory, positions it on the screen, sets its font to Geneva 12, then makes it visible. Note well: it's a good idea to use dialog and window templates that come up invisibly. Then you can bring them into memory, pull off any adjustments in private, and use *ShowWindow* to make them appear.

dealWithDialogItem—Just a big switch statement to case out on the button that got clicked in the main event loop. The top layers of a Mac

application are usually filled with such switch statements as the program zeroes in on exactly what kind of event occurred and what to do about it. Note how the *quitItem* button controls the main event loop via the global Boolean variable *finished*.

The following nine routines deal with the clicks of specific buttons:

doOrwellItem—The *orwellItem* button stays highlighted while the *ronItem* button fades in and out.

doSnapshotItem—The *doSnapshotItem* button lets you take action pictures of the demo program via a call to a Camera desk accessory. If you don't have such a DA in your system file, the *OpenDeskAcc* call returns without crashing.

doMushroomItem—Similar to *doOrwellItem*. This time the *bumperStickersItem* fades in and out of view.

doOpenItem—Calls on the standard file-opening routine, then does nothing with the routine's result.

doSaveAsItem—Calls on the standard file-saving routine, then does nothing with the routine's result.

doFlipItem—Takes a list of content-changing buttons, then runs them through a little animation routine by turning highlighting on and off.

doSomeOffItem—Takes a list of buttons and makes them inactive.

doSomeOnItem—Takes the same list of buttons passed to *doSomeOffItem* and makes them active.

doCopyrightItem—Brings up a modal dialog that expresses the author's interest in legal protection for works of art.

figureCenteredRectTLC—I don't know about you, but I go nuts over programs that don't know how to position things on different-size screens. This little routine shows how simple it is to be tidy.

To be continued next month.

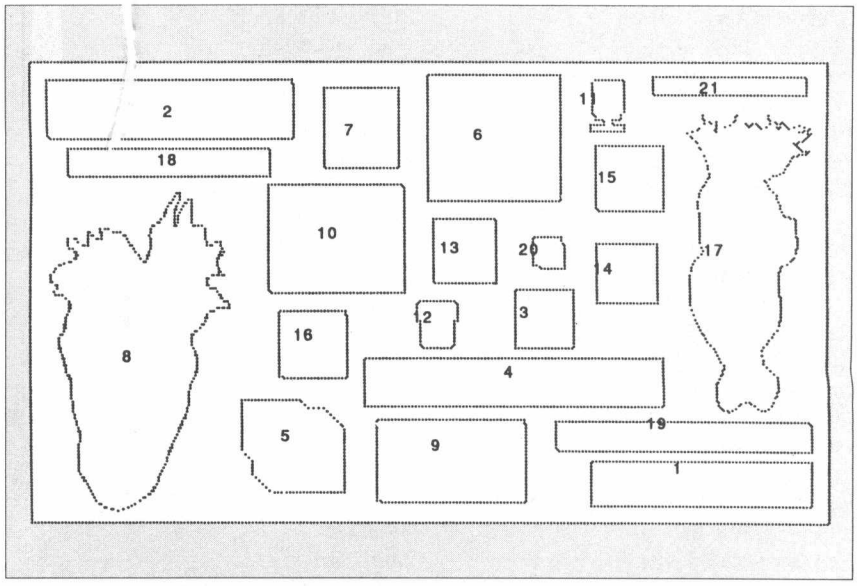


Figure 7: *DITL item numbers for each of the controls in custom control demo's main modal dialog*

TO THE MACS

Wrap Up

Special thanks go to the following for thoughts and actions that made this month's column possible: Tom Atkinson of Orchard Computer, Cynthia Bruschi of ICOM Simulations, Dan Cochran of Apple, Doreen Duplin of Think Technologies, Bruce Hammond of Starpoint Software, Michael Kahl of Think Technologies, Jerzy Lewak of Paragon Concepts, John Mitchell of Apple, David Perlman of Action Graphics, Andrew Singer of Think Technologies, Nathan Slemmer of Interstate Computer Bank, Tyler Sperry of *DDJ*, Mike Swaine of *DDJ*, Levi Thomas, and Dan Weston of Nerdworks.

This is the first of my *DDJ* Mac columns. Feedback pro and con will be much appreciated; my access information is at the end of the column. Hot tips, keen insights, funny problems, and review copies of books and software are also solicited.

Next month for sure: assembly-language source for *rectCDEF* along with copious explanation and the rest of custom control demo's resources.

Next month maybe (depending on time, space, and circumstance): hypertalk secrets, living with multiFinder, macdraw with a brain, parasitic desk accessories, talks with various programming luminaries, and Microsoft madness revealed.

Stan Krute, when not serving as DDJ's new Mac columnist, is an artist, programmer, writer, and teacher. You can reach him via MCI

Vendors

APDA
Apple Programmer's and Developer's Association
290 S.W. 43rd St.
Renton, WA 98055
(800) 426-3667
In WA (800) 527-7562
In Canada (800) 237-4644
(206) 251-5222

Apple Certified Developer Program
Developer Programs
Apple Computer Inc.
20525 Mariani Ave.
Mailstop 27-W
Cupertino, CA 95014
(408) 996-1010

Computer Literacy Bookshops
2590 N. First St.
San Jose, CA 95131
(408) 435-1118
(seven days a week)

Computerware
350 Cambridge Ave.
Palo Alto, CA 94306
(800) 235-1155
In CA (800) 323-1133

Consulair 68000 Development System
Consulair
140 Campo Dr.
Portola Valley, CA 94025
(415) 851-3272
Reader Service No. 29

Enhance Expansion Board
SuperMac Technology
295 North Bernardo
Mountain View, CA 94043
(415) 964-8884
Reader Service No. 30

Fedit Plus
MacMaster Systems
108 E. Fremont Ave., Ste. 37
Sunnyvale, CA 94087
(408) 773-9834
Reader Service No. 31

Fry's Electronics
541 Lakeside Dr.
Sunnyvale, CA 94086
(408) 662-3566

Lightspeed C
Think Technologies
135 South Rd.
Bedford, MA 01730
(800) 648-4465
(617) 275-4800
Reader Service No. 32

MacNosy
Jasik Designs
343 Trenton Wy.
Menlo Park, CA 94025
(415) 322-1386
Reader Service No. 33

MacTutor
MacTutor
P.O. Box 400
Placentia, CA 92670
(714) 630-3730
Reader Service No. 34

QUED/M 2.04
Paragon Concepts Inc.
4954 Sun Valley Rd.
Del Mar, CA 94014
(619) 481-1477
Reader Service No. 35

ResEdit
Available through APDA (see above) or via one of the many on-line services.

Signetics Corp.
Publication Services
Mailstop 27
P.O. Box 3409
Sunnyvale, CA 94088-3409
(408) 991-3620

TMON 2.8
ICOM Simulations Inc.
648 S. Wheeling Rd.
Wheeling, IL 60090
(312) 520-4440
Reader Service No. 36

Mail, Delphi (STANKRUTE), and CompuServe (73137,2121) and also by mail at 18617 Camp Creek Rd., Hornbrook, CA 96044—eds.

Bibliography

Apple Computer Inc. *Inside Macintosh*, 4 vols. Reading, Mass.: Addison-Wesley, 1985–1986.

Harbison, Samuel P.; and Steele, Guy L., Jr. *C: A Reference Manual* (2d ed.). Englewood Cliffs, N.J.: Prentice-Hall, 1987.

Knaster, Scott. *How to Write Macintosh Software*. Hasbrouck Heights, N.J.: Hayden, 1986.

Signetics Corp. *Signetics S68000 User's Guide*. Sunnyvale, Calif.: Signetics Corp., 1983.

Smith, David E. ed. *The Best of MacTutor, Volume 1*. Placentia, Calif.: MacTutor, 1986.

Smith, David E. ed. *The Complete MacTutor, Volume 2*. Placentia, Calif.: MacTutor, 1987.

Weston, Dan. *The Complete Book of Macintosh Assembly Language Programming*, 2 vols. Glenview, Ill.: Scott, Foresman, 1986.

Availability

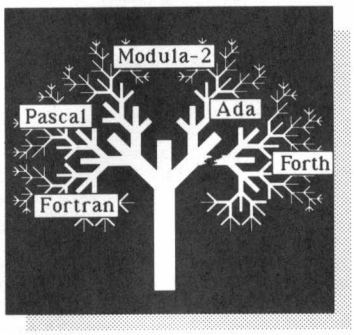
All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 54.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Object-Oriented Programming in Pascal



Object-oriented programming has become quite popular in the last few years. One aspect of its growing acceptance is the increasing number of structured and AI language implementations that support this new paradigm—for example, there are object-oriented implementations in C, Pascal, LISP, Logo, and Forth. In recognition of the magazine's focus this month on the 68000 line, I'll look at a new object-oriented Pascal available for the Macintosh.

The strength and attraction of object-oriented programming is that it offers you both a new concept (or technique, if you like) for modular coding and a method for efficiently describing a hierarchy of data structures (because redundant components need not be redeclared).

Objects in Pascal can be regarded as highly evolved record structures. Objects declare instance variables (similar to fields of Pascal records) and instance methods, or methods for short. These methods are routines that manipulate the objects. Consider the following object-type declaration:

```
TYPE TRectangle = OBJECT
  { instance variables }
  Length,
  Width : real;
  { instance methods }
  FUNCTION Area : real;
  FUNCTION Circumf : real;
  FUNCTION Diagonal : real;
END;
```

by Namir Clement Shammas

which declares a class of objects named *TRectangle* that contains two instance variables—namely, *Length* and *Width*. Three methods are supplied to calculate the rectangle's area, circumference, and length of diagonal. The declared methods (all of which are functions in the preceding example) are implicitly *FORWARD* declarations of the routines' headings. The actual code is placed after all the other declarations.

To use the *TRectangle* object type, I must declare a variable and then allocate its dynamic memory using the predefined *NEW( )* procedure. Accessing the fields and routines employs the familiar dot notation used with Pascal records. Consider the following code fragment:

```
VAR Rect : TRectangle;
  A : real;

BEGIN
  NEW(Rect);
  Rect.Length := 8.0;
  Rect.Width := 5.0;
  A := Rect.Area;
```

In this code the name of the object is used in conjunction with all the object's instance variables and methods.

A rectangular parallelepiped is a 3-D rectangle that can be regarded as extending the 2-D "parent" shape into a third dimension. This means the parallelepiped shares the same features as the 2-D rectangle and adds a few new ones. To enable the parallelepiped objects to inherit the features of the 2-D rectangle, I declare the following:

```
TYPE TSolid = OBJECT(TRectangle)
  height : real;
  FUNCTION Volume : real;
  FUNCTION Space_Diag : real;
END;
```

This declaration states that the object type *TSolid* is a child of the object type *TRectangle*. As a child object, it is able to inherit all the instance variables and methods of its parent object:

```
VAR Solid : TSolid;
  A, V : real;

BEGIN
  NEW(Solid);
  Solid.Length := 8.0;
  Solid.Width := 5.0;
  Solid.Height := 7.0;
  A := Solid.Area;
  V := Solid.Volume;
```

Notice how the variable *Solid* (of type *TSolid*) is able to access the instance variables and methods of the *TRectangle* object type directly. Unlike nested record structures in Pascal, nested objects can make direct reference to the ancestor's variables and methods.

In the preceding discussion, I pointed out that the class of objects *TSolid* is able to inherit components from its parent object. Object-oriented Pascal provides a convenient mechanism to enable a new subclass of objects to define its own version of certain inherited methods. Such methods offer either additional or alternate ways of manipulating an object.

Consider, for example, the case of making an alternate definition of the object type *TSolid*. I want to rename the method *Space_Diag* as *Diagonal*. Because the *Diagonal* name is also used by a method in object *TRectangle*, I must resolve the conflict I have just created. The keyword *OVERRIDE* is used to override inherited versions, so my new alternate declaration for the *TSolid* object type is:

```
TYPE TSolid = OBJECT(TRectangle)
  height : real;
  FUNCTION Volume : real;
```

```
  FUNCTION Diagonal : real; OVER-
                              RIDE;
END;
```

The *OVERRIDE* keyword is placed after the sought method, followed by a semicolon. To revert to the method inherited from a parent object type, instead of the overridden method, the keyword *INHERITED* must be placed before the method in question.

In certain cases it may be necessary to make a reference to an object associated with a method. Object types are similar to pointers, and hence they must be dynamically allocated. A problem arises because methods cannot allocate the objects they manipulate. To solve this dilemma, object-oriented Pascal provides a special identifier, *SELF*. Using *SELF*, you can make references to the object that is yet to be created—for example, the code for method *Volume* can be written as:

```
FUNCTION Volume : real;
BEGIN
  Volume := SELF.Length *
      SELF.Width *
      SELF.Height;
END;
```

TML Pascal for the Mac, the only object-oriented Pascal implementation I've seen, allows you to omit the *SELF* reference.

A Sample Program

Object-oriented programming can be applied across the board, including with basic data structures. Listing One, page 66, shows a complete TML Pascal program that implements simple objects to represent various types of simple numeric stacks.

The first object defined, *TStack*, contains instance variables and methods used by all the other object types. In this case, the instance variables include the stack height and a Boolean flag used to indicate errors (in the program I use it to indicate that an attempt was made to pop an empty stack; you can also use the flag to indicate an attempt to divide by 0). The methods associated with *TStack* objects initialize, increment, and decrement the stack height.

```
CONST MAX_MENU = 20;

TYPE STRING80 = STRING[80];
     String_Array = ARRAY [0..MAX+MENU] OF STRING80;
     Menu_Range = 0..MAX_MENU;

TMenu = OBJECT
     { declare instance variables }
     Menu_Options : String_Array;
     Num_Options,
     Menu_Choice : Menu_Range;
END;

TItem_Menu = OBJECT(TMenu)
    PROCEDURE Display_Menu;
    FUNCTION Get_Choice : Menu_Range;
END;

TControl_Item_Menu = OBJECT(TMenu);
    Current_Level : Menu_Range;
    PROCEDURE Display_Menu;
    FUNCTION Get_Choice : Menu_Range;
END;

TMain_Pull_Down = OBJECT(TMenu)
    PROCEDURE Display_Menu;
    FUNCTION Get_Choice : Menu_Range;
END;

TPull_Down = OBJECT(TMenu)
    Hot_Key_Char : ARRAY [0.MAX_MENU] OF CHAR;
    Location : ARRAY [0.MAX_MENU] OF INTEGER;
    Attribute : ARRAY [0.MAX_MENU] OF BYTE;
    Active : ARRAY [0.MAX_MENU] OF BOOLEAN;
    PROCEDURE Display_Menu;
    FUNCTION Get_Choice : Menu_Range;
END;
```

Example 1: *Declaration of object types for various kinds of menus*

(continued from page 109)

The second object type, *TRealStack*, is a child of object *TStack*. It defines the stack as a four-element, real-type array. The methods associated with this object type are *Push*, *Pop*, and *Add*. (I've omitted other math-related routines to keep the listing short.) The third type, *THPStack*, is a child of the object type *TRealStack*. This new object type defines the instance variable *LastX* to store the value of the first array element when a stack addition is performed. This also dictates that an overridden *Add* method be defined.

Finally, the fourth object type, *TIntStack*, is a descendant of object *TStack* that implements an integer-type version of object *TRealStack*. The main code portion exercises the methods defined with the objects.

Menus As Objects

Objects find applications in other data structures, such as lists, arrays, and matrices. They can also be applied to data structures representing screens, windows, and menus. Example 1, page 109, shows the declaration for object types representing different kinds of menus.

The *TMenu* object type defines instance variables that perform the following:

- Tackle the text for menu options.
- Store the number of actual options available.
- Store the number of the option selected (the 0th option is used for exiting from the menu).

Object-oriented Pascal limits objects to single inheritance

TItem_Menu is a menu object type that builds on *TMenu* simply by adding two methods: one to display the menu and another to return the selected choice. The object type *TControl_Item_Menu* is a modified version of *TItem_Menu* that allows you to implement progressive menus that gradually reveal more options. These menus display an itemized menu that contains only the options available to you at the current level.

The *TMain_Pull_Down* object type uses the same instance variables of object type *TMenu*; however, it implements its own version of the methods to display the main menu of a pull-down menu system. The individual options of a pull-down menu are defined by the object type *TPull_Down*. This type declares an additional number of instance variables that are arrays that serve to:

- Define the hot-key characters.
- Locate where the hot-key characters are displayed and their accompanying display attributes.

- Define Boolean flags to indicate whether an option is active or passive.

Inheritance Problems

Object-oriented Pascal limits objects to single inheritance: one object type has at most one parent object type. Other languages and implementations, such as Smalltalk, Object Logo, and ExperCommon LISP, support multiple inheritance. I'd like to see multiple inheritance implemented in future versions of object-oriented Pascal because it offers the flexibility and power to model real-world objects realistically. This new level of sophistication generates new problems, though.

Among the first problems to resolve is the question of inheriting instance variables and methods that are present in the ancestor objects. The solution involves assigning influence levels, or priorities, to ancestor objects and providing the ability to override them. I suggest some alternate rules for resolving inheritance conflicts by assigning influence levels as follows:

1. The first ancestor object mentioned is the dominant parent, and all other ancestor objects are of equal importance. In the following example:

```
TYPE Car = OBJECT(Make, Engine, Body)
```

the ancestor object *Make* has the greater influence concerning conflicting instance variables and methods and the objects *Engine* and *Body* have the same influence. This syntax assumes that there are no conflicts pending between the last two object types.

2. The order of listing the ancestor objects indicates their influence levels. The following object heading declaration illustrates this syntax:

```
TYPE Car = OBJECT(Make, Engine, Body)
```

Here, the *Make* and *Body* object types have the strongest and weakest influences, respectively.

3. The list of ancestor object types is partitioned into two sublists using a special character—say, the bar symbol. The first sublist has its ancestor object types listed in decreasing influence; the second has the rest of the ancestor object types on equal footing. Consider the heading:

```
TYPE Car = OBJECT(Make, Engine
                  | Body, Doors)
```

The first sublist contains object types *Make* and *Engine*, and the second sublist contains *Body* and *Doors*. The object type *Make* has the dominant influence, followed by *Engine*. The *Body* and *Doors* object types have an equal influence, which is weaker than that of the first two. This syntax also assumes that there are no conflicts pending between the last two object types.

4. Influence levels are explicitly assigned to all ancestor object types using the syntax *<object type> = <unsigned integer constant>*. The integers used need not be in any particular sequence, as shown in the following example:

```
TYPE Car = OBJECT(Make = 100,
                  Engine = 1, Body = 40)
```

This object type declaration indicates that the object type *Make* has

the highest influence (by virtue of the number assigned to it and not its position in the list of object types). By contrast, the *Engine* object type is the least influential.

The first and third alternatives contain a common weak point: they may be unable to resolve all the inheritance conflicts. The second and fourth suggested syntaxes are conflict-proof. Nevertheless, you need a mechanism to arbitrate or override inheritance rules explicitly. I suggest using the keyword *FROM* to indicate the parent object type supplying the particular attribute, as shown by the following general syntax:

<variable> : data <type> FROM <object type>
<method> FROM <object type>

Tracking down inherited variables and methods in multiple inheritance is much more complex than in single inheritance. A single link in the latter is replaced by a complex search graph in multiple inheritance. The increased real time for processing multiple inheritance could be compensated for, however, by using fast CPUs and electronic disks, which would keep the overall compilation time relatively short.

A Final Note

Writing this column for the last few years has been fun, but it has also required a great deal of writing time. Other obligations, including the job of editor of M&T's *Turbo Tech Report*, have made it necessary for me to pass this column's duties on to another member of the *DDJ* family. Although I will continue to write an occasional article for *DDJ*, this month marks my last Structured Programming column. Thank you for your support, and take care of yourselves.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Bibliography

Cox, Brad J. *Object-Oriented Programming: An Evolutionary Approach*. Reading, Mass.: Addison-Wesley, 1987.

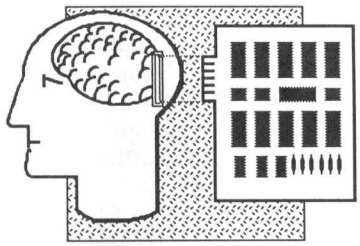
Schmuker, Kurt J. *Object-Oriented Programming for the Macintosh*. Hasbrouck Heights, N.J.: Hayden, 1986.

DDJ

(Listing begins on page 66.)

Vote for your favorite feature/article
Circle Reader Service **No. 5.**

Actor Does More Than Windows



This month I'll be looking at the latest version of The Whitewater Group's object-oriented language Actor. The Whitewater Group has been promoting this language simply as an effective tool for rapid development under Microsoft Windows. It is certainly that. But Actor is also the first object-oriented programming tool specifically designed for writing applications that run under Windows. That fact and The Whitewater Group's intention to port Actor for other windowing systems make it an interesting tool for developing precisely the AI applications that the object-oriented approach is good at while easing the burden of developing for different machines. In this column I'll describe how the latest version (1.1) of Actor implements some of the Microsoft Windows constructs and will examine it from that perspective.

There are so many similarities between Smalltalk and Actor that you may well ask why anyone would develop a new language when implementations of Smalltalk already exist. There are two reasons. First, Actor was designed from the start to be compatible with commercial personal computer environments. Hence its built-in compatibility with Microsoft Windows today and The Whitewater Group's plans to allow window environment programming with Actor across windowing environments tomorrow. Second, the syntax of Actor has been designed to be familiar to C and Pascal programmers. The Whitewater Group's intent was to incorporate all that was good about Smalltalk in a package tailored to the needs of present-day personal computer programmers.

by Ernest R. Tello

To give you a feeling for programming in Actor, I have sprinkled some sample programs through this column as Examples 1 through 4, pages 116, 118, and 125. Three of these are familiar benchmark programs for generating prime and Fibonacci numbers and the Tower of Hanoi and the fourth contains Actor author Chuck Duff's extensions for list processing.

The Actor Desktop

Basically, Actor runs on top of MS-Windows and uses the Windows facilities to implement objects that allow an interactive, windowing environment for development. As with MS-Windows, you can theoretically get along without a mouse, but you wouldn't want to.

The main items used on the Actor desktop are the workspace windows, browsers, and inspectors, which are all modeled on the Smalltalk originals.

When the system first comes up, it shows a workspace window with two rows of command options along the top bar of the window. The commands include *File*, *Edit*, *Doit!*, *Browse!*, *Inspect!*, *Show Room!*, and *Templates*.

The editing area of a workspace window behaves just like an interpreter does. If you type in an expression and then a carriage return, Actor will attempt to compile and execute it. If there is a body of text already in an editing window, then highlighting a portion of that code and clicking on *Doit!* will result in that portion of code being compiled and executed. The *Browse!* and *Inspect!* options result in a new one of these tools opening like a pop-up window on the Actor desktop.

Inspectors are used for focusing on a particular object. You use them to examine the contents of objects in detail as well as to make modifications to them. The upper-left pane of an inspector window contains a list box that displays all the instance variables of an object. Clicking the mouse on any of the items in the scrollable list of variables results in its value being displayed in the bottom pane of the inspector—its edit window. Both class objects and their instances can be accessed using inspectors.

A browser provides a similar function to that of an inspector, but instead of providing an interactive window on a single object, it does this for the entire system of classes. Its scrollable list box contains a list of all the classes currently in the Actor class hierarchy. The right-hand list box contains the methods for the current class. By first selecting a class and then a method in that class, you may access the code in the bottom window and edit it. An Options selection on the browser menu bar allows you to choose whether the classes are listed in hierarchical or alphabetical order.

A Class Act

Actor comes with a large class library of ready-made code that can be used for building applications quickly, once you bridge the learning curve of using the system and knowing what's there. Here is a partial list of the classes used for data structures and the graphics facilities subsumed under them:

```
Object
   Collection
      Indexed Collection
         Array
            Function
            OrderedCollection
               SortedCollection
```

(continued from page 114)

TextCollection
ByteCollection
String
Symbol
Struct
DosStruct
GraphicsObject
Polygon
Rect
Ellipse
RndRect
Proc

Actor Syntax

As with languages such as C and Pascal, Actor encloses arguments to a method in parentheses immediately following the method's name or selector. And like Pascal and Smalltalk, variable assignments are made using the colon-equal (:=) symbol. Also as in Smalltalk, the way you create new instances is by sending the *new* message to a class and assigning this new instance a name. So, for example, you could create an instance of the *Turtle* class by saying:

```
inherit(Object, #Sieve, nil, nil,nil);!!

now(Sieve);!!

/* Returns the number of prime numbers between 0 and cnt,
inclusive. */
Def sieve(self, cnt | flags, count, c)
{       c := cnt + 1;
        flags := new(Array, c);
        fill(flags, true);
        count := 1;
        do( over(2, c),
                { using(i | triple)
                        if flags[i]
                        then triple := i*3;
                                if triple < cnt
                                then  do( overBy(triple-1, c,
i+i-1),
                                        { using(j) flags[j] := nil ));
                                endif;
                                count := count + 1;
                        endif;
                });
        ^count;
}!!

Actor[#Sam] := new(Sieve)!!
/* To run type: sieve(Sam, 100) */!!
```

Example 1: *Eratosthenes' Sieve benchmark*

```
now(Int)!!

/* Recursive way of finding the nth Fibonacci term.  Note that
this way of finding the Fibonacci terms is very inefficient
because each message "spawns" two recursive messages.  */

Def fib(self)
{ if self < 3
 then ^1
 endif;  ^fib(self - 1) + fib(self - 2);
}!!

/* Iterative way of finding the nth Fibonacci term. */

Def fib2(self | term, term1Before, term2Before)
{ if self < 3
  then ^1
  else term := 2; term1Before := 1; term2Before := 1;
    do(new(Interval, 3, self + 1, 1),
      {using(i) term := term1Before + term2Before;
       term2Before := term1Before;
       term1Before := term;
      });
    ^term
  endif;
}!!
```

Example 2: *Fibonacci program*

(continued from page 116)

```
Barney := new(Turtle);
```

In general, sending messages in Actor is like passing arguments but in reverse. The object to which the message is sent is treated as an argument. Once you have created the turtle *Barney*, you can get him to do your bidding by sending various messages that he can recognize. Like any authorized turtle, *Barney* knows that the message *r* means turn right, *l* turn left, *f* move forward, and *b* move backward. He also knows that *down* means to put his tail to the ground for drawing purposes and *up* means to pick it up again.

So, if you wanted *Barney* to perform a turtle walk in the shape of a square, the messages you would send him would be:

```
down(Barney);
f(Barney, 10);
r(Barney, 90);
f(Barney, 10);
r(Barney, 90);
f(Barney, 10);
r(Barney, 90);
f(Barney, 10);
up(Barney);
```

In Actor, the keyword *Def* is used to define methods. So, if you wanted to teach not only *Barney* but also all authorized turtles the new message *walkSquare*, you would define the method:

```
Def walkSquare(self, size)
{ down(self);
  f(self,size);
  r(self, 90);
  f(self, size);
  r(self, 90);
  f(self, size);
  r(self, 90);
  f(self, size);
  up(self);
};
```

Henceforth, to get *Barney* or any of his relatives to perform this maneuver, all you would have to do is to say:

```
walkSquare(Barney, 10);
```

The more astute turtle watchers have probably noticed that this turtle walk is only one orientation for this type of maneuver. There is a species of turtle, admittedly rare, that instinctively will do its square-walking counterclockwise. Fortunately, object-oriented systems such as Actor provide a way of mirroring this little sidelight of natural history. What you can do to cover this complexity is to define a new class of turtle called *CounterTurtle* and provide a *walkSquare* method for turtles of this species that does the counterclockwise variant of the standard turtle square walk. Coincidentally, this also provides me with the opportunity of illustrating how new classes are defined in Actor.

The way you usually create new classes in Actor is from within a browser, so I'll do it that way first. Very simply, you first select the class *Turtle* from the class list. Then you go to the Options pull-down menu and select *Make Descendant*. A pop-up window then opens that serves as a template for creating the new class. In this case, let's enter *CounterTurtle* as the name of the class. Now you just click on the Accept button, and the system will create this new class and its name will be added to the class list and become part of the Actor class hierarchy.

The other way to create new classes, which is what the browser is actually doing, is to write the code for it directly. The *inherit* statement is used for this. So, you could write:

```
inherit(Turtle, #CounterTurtle) !!
```

One of the most attractive things about the Actor system is that it provides built-in classes for making the use of the Microsoft Windows user interface easier. Three main classes are concerned with this: *Window*, *Control*, and *ModalDialog*. First let's look at the *Window* class and its descendants. Here is an outline of this branch of the class hierarchy:

```
Object
  Window
    PopupWindow
      ToolWindow
        Browser
        Inspector
    TextWindow
      EditWindow
        WorkEdit
          BrowEdit
          FileWindow
          Workspace
      ScanWindow
      WorkWindow
```

Though relatively large, *Window* is just a formal or abstract class. This means that it implements the methods that will be used by the sub-

```
/*  ref.  Byte August, 86  p. 146   cbd  8.13.86 */!!

inherit(Object, #TowerOfHanoi, nil, nil, nil);!!

now(TowerOfHanoi);!!

Def moveTower(self, height, from, to, use)
{  if height > 0
   then
     moveTower(self, height - 1, from, use, to);
     moveTower(self, height - 1, use, to, from);
   endif;
}!!

Def moveTower2(self, height, from, to, use)
{  if height > 0
   then
     moveTower(self:TowerOfHanoi, height - 1, from, use, to);
     moveTower(self:TowerOfHanoi, height - 1, use, to, from);
   endif;
}!!

Actor[#Hanoi] := new(TowerOfHanoi);!!

/* Example solves runs the Tower of Hanoi problem */!!
moveTower(Hanoi, 3, 1, 3, 2);!!
```

Example 3: *The Tower of Hanoi*

classes that implement the specialized windows that actually get instantiated and used. In particular, *Window* implements the routines that communicate with MS-Windows.

The *TextWindow* class is one of the simplest descendants of *Window* that you can get to actually do real things. This class allows you to create tiled windows that can print text. It does this with the *printString* and *printChar* methods, which call the *Textout GDI* (Graphics Display Interface) function in MS-Windows.

Often you will want text windows that can do more that just show text—that can allow you to go in and edit that text. The subclass of *TextWindow* called *EditWindow* provides the code that supports this editing ability. The *WorkEdit* class takes this one step further by allowing you to create windows that can not only edit but can also enter Actor language statements to be evaluated. The three subclasses of *WorkEdit* provide the types of windows that are like those most often used—browsers, file browsers, and general-purpose workspace windows. The main difference, though, is that, like their ancestor *TextWindow*, these are tiled windows.

The windows most often used in Actor come from another branch of the tree that is implemented with the class *PopupWindow*. The windows you get by instantiating *PopupWindow* are the familiar layered windows that stack up on top of one another. Unlike the tiled windows, however, they do not permit you to zoom or contract them down to an icon. As is dictated by MS-Windows, pop-up windows have to have a "parent" text window. When this parent window is contracted into an icon, then the pop-up windows associated with it temporarily become invisible.

Controls and Dialog Boxes

As in MS-Windows, in Actor a control is a special type of window that is used for routine input and output in a user interface. Examples of controls include things like buttons, list boxes, and scroll bars. The branch of the class tree concerned with controls looks like this:

```
Object
   Control
      Button
      ListBox
         ClassList
      Scrollbar
```

Like *Window*, the *Control* class in Actor is just a formal one, and its subclasses are the ones that are actually instantiated in applications. Controls are handled in Actor in an almost identical way to windows. New instances are created by sending the *new* message, and they are displayed by sending the *show* message.

I'll describe one other class—the *ModalDialog* class and its subclasses.

```
Object
   ModalDialog
      ClassDialog
      DebugDialog
      DirtyCLD
      FileDialog
      InputDialog
```

Modal dialog boxes resemble pop-up windows in that they stack on top of other windows and they need a parent window with which they are associated. Like *Window* and *Control*, *ModalDialog* is basically an abstract class that implements code intended for use by its descendants. The *FileDialog* class in Actor is used to create the dialog boxes that routinely appear in MS-Windows when you load a file by using a pull-down menu. The *ClassDialog* class is for dialog objects used when a class is being edited or created with a browser.

Using Actor for AI

For AI applications, as well as many other types of application, processing linked lists is essential. How do you go about doing that in Actor? One way might be to work with the *OrderedCollection* class and add subclasses to it with the necessary methods defined for list processing.

Example 4, a demo provided with the current release of Actor, offers another approach, however. The new class *ListNode* is defined as a subclass of the *Collection* class. The

(continued from page 120)

methods *append*, *do*, *isAtom*, *printOn*, and *rPrintOn* are defined for this new class. New methods, including *isAtom*, *rPrintOn*, and *cons*, are also defined for the root class, *Object*. If you inspect the code for *cons*, you will see that there is a routine for sending the message *new* to the *ListNode* class and creating a new instance of it. Obviously, this is only the rudimentary beginning of what a functional list-processing class would encompass.

As far as the suitability of Actor for AI applications is concerned, the same limitations that apply to Smalltalk apply here. As compared with LISP and PROLOG, Smalltalk and Actor are relatively low-level languages. They are suitable for developing AI applications, but many additional high-level methods and classes have to be written from scratch just to get started.

The structures used by the *OrderedCollection* class differ significantly from dynamically modifiable linked lists. In the terminology of object-oriented programming, ordered collections are fixed collections that are nevertheless "growable." This means that when you create an *OrderedCollection*, you must create one with a maximum number of elements. If the elements already stored in the collection have not yet reached the maximum, it is easy to add new elements to the beginning or end of the list. When the maximum is reached, and you need more, you must send the *grow* message to the collection. What really happens when you do this is that a new array of the needed size is created and the elements of the old array are copied into it.

Debugging

With Version 1.1 a new debugger has been added to Actor. Currently, both a low-level and high-level debugger are provided, but the low-level debugger is not formally supported and may disappear in later releases of the Actor system.

Routine errors in code evaluated by Actor result in a dialog box that contains a stack history up to the point of the error. The dialog box usually also contains a message that diagnoses the type of error. If you wish, when a dialog box is open because of an error, you can click on the Debug button and cause a Debug window to open.

This is a versatile debugging tool that combines some of the features of a browser and some of those of an inspector as well as the ability to change any of the values associated with a method. With it you can also resume processing on the fly immediately after an error has been fixed.

Conclusions

On the whole, I find Actor to be a thorough implementation of a full programming system with an excellent set of demo programs and helpfully written documentation and tutorials. The ideal users of Actor, as I see it, would be programmers who have already had some exposure to Smalltalk and need to prototype something quickly to run in the MS-Windows environment. For purposes such as these, it is hard to beat.

One thing about the implementation of Actor I dislike, though, is the absence of a facility for multiple inheritance. With systems intended for real-world applications, multiple inheritance should be a standard feature because in the real world many things fulfill multiple roles and multiple functions. Multiple inheritance provides a ready way of handling this in an explicit way.

Vendors of object-oriented tools who fail to include multiple inheritance typically say that they do not want to make the system too complex for users or that none of their customers have requested it. I have not found either of these explanations at all convincing. I have had no difficulty in using multiple inheritance in systems that have it and can't imagine trying to build a serious object-oriented application without it.

In response to this, it might be said that you can still create classes of the same definition in a system without multiple inheritance the hard way simply by defining them to be exactly what you want. My feeling about this is that it may be true in theory but it tends to be something that is never done in practice. In the two years or so that I have worked with object-oriented programming systems, I have never done this unless the system provided for multiple inheritance, in which case it becomes a routine practice.

ARTIFICIAL INTELLIGENCE

The reason is the same one that makes interactive systems such as Actor significant and not just a mere convenience. The more you make basic things easy to do, the more you tend to launch out into the more difficult areas creatively, trying things that otherwise you might not have tried. If you are a user of an object-oriented programming tool that lacks this feature, I strongly recommend that you pester the vendor to put it at the top of the list for new features. I feel almost certain that you will not regret the results of having exercised your prerogative as a customer.

Version 1.1 of Actor differs from 1.0 in two main ways. First of all, there is more space—an additional 70K—for compiling applications. Second, it is fully compatible with Windows II, though it does not fully support all of Windows II's facilities. A future release of Actor, though, will actually support programming with the new features of Windows II. Needless to say, the implementations of Actor that will use the full features of Windows II and the OS/2 Presentation Manager will ultimately determine the fate of this product. But if the current implementation is any indication, the future versions should be of very high quality indeed.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Vendor

Actor
The Whitewater Group
Technology Innovation Center
906 University Pl.
Evanston, IL 60201
(312) 491-2370
Reader Service No. 28

```
 C.B.Duff  7.13.86
 (c) Copyright, 1986
 The Whitewater Group
 Technology Innovation Center
 906 University Place
 Evanston, Il. 60201  (312) 491-2370       */


now(NilClass);!!
/*  append nil to a node  */
Def append(self, aNode)
{ ^aNode }!!

Def cons(self, aNode)
{ ^aNode }!!

Def rPrintOn(self, aStrm)
{ printOn('[', aStrm);
}!!

inherit(Collection, #ListNode, #(left right), nil,nil);!!

now(ListNode);!!

Def append(self, aNode)
{ ^cons( left, append(right, aNode));
}!!

Def do(self, aBlock)
{ if isAtom(left)
 then eval(aBlock, left);
 else do(left, aBlock);
 endif;
 if right
 then do(right, aBlock);
 endif;
}!!

Def isAtom(self)
{ ^nil }!!

Def printOn(self, aStrm)
{ printOn('[', aStrm);
 printOn(left, aStrm);
 rPrintOn(right, aStrm);
}!!

Def rPrintOn(self, aStrm)
{ printOn(' ', aStrm);  printOn(left, aStrm);
 rPrintOn(right, aStrm);
 printOn(']', aStrm);
}!!

now(Object);!!

Def isAtom(self)
{ }!!

Def rPrintOn(self, aStrm)
{ printOn('.', aStrm);
 printOn(' ', aStrm);
 printOn(self, aStrm);
 printOn(']', aStrm);
}!!

Def cons(self, aNode | newNode)
{ if isAtom(aNode)
 then newNode := new(ListNode);
 else newNode := copy(aNode);
 endif;
 newNode.left := self;
 newNode.right := aNode;
 ^newNode;
}!!
```

Example 4: *List-handling support in Actor*

(continued from page 12)

is a defining word *UNITS*:

```
: UNITS CREATE SWAP , , DOES>
                        D@ */;
```

Then you would have what appears in Example 1, page 12. This is an improvement, but Forth permits a simple addition that allows conversion back to the input units when outputting (see Example 2, page 12).

Obvious modifications include use of floating point (to prevent the unintentional truncations of integer arithmetic) and defining a machine-code sequence *<DO-UNITS>* to replace the run-time instructions following *DOES>* if greater speed is needed. (It probably isn't—these definitions are intended to I/O a relatively small set of measurements from the console or a file.)

Finally, let me put in a plug for HS/FORTH (for IBM PCs/clones): Jim Callahan's Harvard Softworks (P.O. Box 69, Springfield, OH 45066) has been extremely supportive with upgrades, fixes, technical advice, and so on. The version of Forth is extremely fast "as is" but comes with all sorts of goodies and tools for using the 8087 chip, string handling, windowing, graphics, sound, and so on. Although the documentation of several years ago was rather cryptic, it is now excellent and includes Kelly and Spies' textbook. I have never seen this product reviewed or mentioned and cannot imagine why, considering its excellence.

Julian V. Noble
105 Powhatan Cir.
Charlottesville, VA 22901

Instruction Timings

Dear *DDJ*,

I hate to be the bearer of bad news, but Tom Disque's "8088 Assembly-Language Programming Techniques" in the July 1987 issue is inaccurate. In several cases Disque presents instruction timings that are not consistent with his assumptions.

In the second paragraph, Disque says, "Please note that all timings are for the 8088 microprocessor and assume that the 4-byte 8088 prefetch queue is empty at the start of execution." Although this is a good assumption because the 8088 is so common and the prefetch queue is always empty except for a few relatively rare circumstances, the timings that Disque gives for his instruction sequences do not correspond to this statement.

Disque claims that the simple sequence:

```
mov ax, cs
mov es, ax
```

takes four cycles to execute. If the prefetch queue is empty, however, the instructions must be fetched first. These two instructions are 4 bytes, and the 8088 requires four clock cycles per memory cycle. This alone comes to 4×4, or 16 clock cycles, just to fetch the instructions, never mind any additional execution time. If the prefetch queue were full, you would have a different story, but Disque assumed that it wasn't.

I suspect that Disque probably meant to do better than this because later he gives the example:

```
shl ax, 1
shl ax, 1
shl ax, 1
shl ax, 1
```

He says: "Faster (8 cycles [plus 8 more to fetch the two extra instructions] ...." I'm not sure what to make of this. Either he meant to assume the prefetch queue was empty or that it was full. In any case, it would take 16 clock cycles to fetch two instructions, not 8 (4 clock cycles per byte times 2 bytes per instruction times 2 instructions).

Let's examine how the 8088 really executes the previous sequence. First, it must fetch the first instruction—eight clocks. It then starts fetching the second instruction while processing the first one. Processing the instruction takes two clocks and fetching the second instruction takes eight. Because eight is more than two, you really have to wait eight clock cycles. At this point the cycle repeats. The total "execution" time is 32 clock cycles.

As you can see, it takes the 8088

much longer to fetch its instructions than to execute them, so the 8088's speed is proportional to the number of bytes it must transfer. This includes instruction bytes as well as data processed by a program. This also means that the 8088's 4-byte queue is almost always empty.

The few exceptions are multiply and divide instructions, which take longer to execute than to fetch. There are also a few esoteric instructions that take longer to execute than to fetch, but their use is so rare that you can forget about them.

Therefore, to time 8088 code that does not include multiply or divide instructions, all you have to do is add the number of bytes of instruction that must be fetched to the number of bytes of data transferred in memory. Once you have the total number of reads and writes to memory, you multiply by four clocks per fetch. You need no instruction timing tables or photographic memory. All you need to be able to do is count and multiply by 4 (two left shifts in binary).

This same concept is used to time Z80 code, except for one quirk. The Z80 takes four clocks for the first byte of an instruction fetch and three clocks to fetch the data.

What about the 8086? The 8086 can fetch instructions faster than the 8088 can but is still basically limited to how fast it can fetch instructions (and read/write to memory). The 80386 should finally provide enough memory width (32 bits) to make a prefetch queue worthwhile.

Disque redeems himself, however, with his Example 8, which demonstrates an excellent, fast, general-purpose, memory move routine. The correction for odd address transfers shows good understanding of that aspect of the 8086 processor.

Good assembly-language programmers such as Disque are rare, but sometimes we all become victims of Intel's overly optimistic instruction timings. Disque's instruction timings sound like they came from Intel.

Lee Pelletier
5 Burley St.
Wenham, MA 01984

Correction

Dear *DDJ*,

I'd like to provide some additional information about the Async AppleTalk article published in the October 1987 issue of *DDJ*.

Async AppleTalk is based on the source code of Apple Computer's AppleTalk driver. The following notice was dropped from the article during the editing process: The AppleTalk protocols and computer programs are licensed from Apple Computer Inc. and AppleTalk, Macintosh, and LaserWriter are trademarks of Apple Computer Inc. Portions copyright (c) 1985 Apple Computer and copyright (c) 1987 the Trustees of Dartmouth College.

Also, the disk being distributed by *DDJ* does not contain the executable desk accessory. This, along with the source files and other debugging tools, is available from the authors. Please send a blank (800K) Macintosh disk in a mailer to me at the address listed below.

Richard Brown
Dartmouth College
Async AppleTalk Distribution
Kiewit Computer Center
Hanover, NH 03755

Stonecutter Error

Dear *DDJ*,

With reference to the cover of your September 1987 issue, the statement *TAKAGUCK* will never be executed because *ZOK-OOT-UK* always sets *GLOOTBLEE* false. The correct statement should be *XOK-OOT-OK*, which uses *GLOOTBLEE* as an error indicator. Please inform your stonecutter to proofread the algorithms before printing.

James Sturdevant
Minneapolis, MN

To List or Not to List

Dear *DDJ*,

In response to your question about source listings in Running Light (November 1987), I personally feel a big value in *DDJ* is the source listings. I use them and I'm sure a lot of the silent majority do too. You're right about a typing drill, so I normally try to download instead to typing. I agree with posting them on CompuServe, and maybe the idea of a BBS run by *DDJ* would be a good addition. Most of the bigger PC publications have established one for their readers.

One different idea I would like to push is the use of Cauzin Softstrips from Cauzin Systems ([203] 573-0150) in each publication, like *PC Magazine* does. Cauzin readers are cheap ($200) and will pay for themselves to readers like me who are far removed from California. I thought you were using Softstrips at one time, and I'm not sure what happened. Anyway, it is a cheap alternative for avid software users like me.

I hope you will consider my suggestion. It would sure be a help to a lot of us.

David Doss
Computer Science Dept.
Illinois State University
133 B Stevenson Hall
Normal, IL 61761

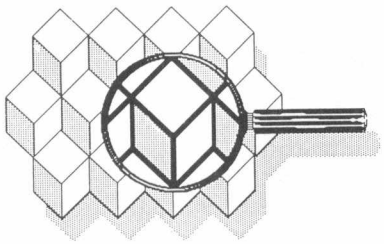
Dear *DDJ*,

I have a few comments regarding your question about the source listings. Keep small (one or two pages) listings. Put long listings on a BBS and make them available on disk. CompuServe is no solution. *PC Magazine*, *Byte*, and *Computer Language* all have great BBSs for downloading. Also, make sure all necessary *include* files are present for C programs. Most published C source code is useless because the author uses routines hidden in header files that are not provided.

Edgar T. Lynk 2G
70 Park Terrace East
New York, NY 10034

DDJ

OF INTEREST



Language Products for the Mac

Two new versions of LPA MacPROLOG are available from **Logic Programming Associates**: the Student Edition uses a built-in incremental compiler and a high-powered declarative graphics environment; the Wizard Edition also has a built-in incremental compiler and high-powered declarative graphics environment along with a new C and Pascal interface, serial I/O, and an optimizing compiler. Both versions require 1 megabyte of memory. The Student Edition costs $275, and the Wizard Edition costs $495. Reader Service No. 16.
Logic Programming Associates
Studio 4
The Royal Victoria Patriotic Building
Trinity Rd.
London SW18 3SX
England
01-8711-2016

Version 2.0 of True BASIC, a language system for the Macintosh and the Macintosh II developed by **True BASIC Inc.**, is now shipping. True BASIC 2.0 offers several new features, including enhancements to the editor; new debugging tools; enhanced speed; and modules, a programming concept found in Modula-2 and Ada. True BASIC supports color graphics for the Mac II as well as the 68881 math coprocessor. Programs written in True BASIC on the Mac can also run on the IBM PC, Amiga, and Atari ST. The retail price of True BASIC 2.0 is $99.95. Owners of previous versions can upgrade to the new version at a cost based on a sliding scale, depending on the date of original purchase. Reader Service No. 17.
True BASIC Inc.
39 South Main St.
Hanover, NH 03755
(604) 643-3882

TurboGeometry Library, a new software tool for Macintosh programmers who write graphics, engineering, educational, and other programs that use geometry, has been released by **Disk Software**. The library comes with source code for more than 150 routines, including routines that find the intersection of lines, polygons, arcs, and planes; determine the coefficients of the equations of lines, circles, arcs, and planes; convert the coefficients of one equation to another; find the distance between points, lines, circles, arcs, and planes; decompose a concave polygon into a series of convex polygons; and more. The library sells for $99.95. Reader Service No. 18.
Disk Software Inc.
2116 East Arapaho, Ste. 487
Richardson, TX 75081
(214) 423-7288

Mac AI

ExperTelligence is now shipping two AI products for the Macintosh: ExperCommonOPS5 and ExperCommon LISP Foreign Function Interface.

Some of the features of ExperCommonOPS5 include a forward-chaining, rule-based, expert system language; a custom tracing mechanism of rule firings and changes to memory; the ability to pause, examine memory, and undo rule firings to revert to an earlier state; conformation to the de facto OPS5 standard described in *Programming Expert Systems Using OPS5* (Brownston et al., Addison-Wesley); complete portability to other implementations of OPS5; facilities to interface with other programming languages and tools; the ability to create complex Macintosh user interfaces for developed expert systems; and integration with the most powerful AI development environments available today. ExperCommonOPS5 runs in both ExperCommon LISP for the Macintosh Plus and SE and ExperCommon LISP II for the Macintosh II, Version 2.2 or later. The retail price is $595.

The ExperCommon LISP Foreign Function Interface is an add-on component of the complete ExperProfessional Development System. Users can create a new version of the ExperCommon LISP language that contains functions written in any Macintosh Programmer's Workshop language: C, Pascal, or assembly language. Once the new version has been created, the foreign functions can be called from ExperCommon LISP in the same way as any other built-in function. The Foreign Function Interface runs in ExperCommon LISP on the Macintosh Plus and the Macintosh SE and runs on the Macintosh II with ExperCommon LISP II. The price is $295. Reader Service No. 19.
ExperTelligence Inc.
559 San Ysidro Rd.
Santa Barbara, CA 93108
(805) 969-7874

Human Intellect Systems has announced Instant-Expert Plus, a Macintosh expert system shell. Instant-Expert Plus uses natural-language rule entry, interactive graphics, variables, and more than 18 different inference engine search strategies. Hardware requirements are a Macintosh 512K with an external drive. Instant-Expert Plus retails for $498. Reader Service No. 20.
Human Intellect Systems
1670 S. Amphlett Blvd., Ste. 326
San Mateo, CA 94402
(415) 571-5939

For the Atari ST and Amiga

A new version of the Metacomco Toolkit is now available from **Metacomco**. The toolkit consists of 11 useful AmigaDOS commands that are stored in the C directory and can be used in the same way as the standard DOS commands. The new version also includes a Unix-based make utility and a touch utility. Commands included in the package are Pipes, Librarian, Disassembler, Auxil-

iary CLI, Mount, Browse, Enlarge, Pack, Unpack, Make, and Touch. Metacomco Toolkit 1.2 works under Versions 1.1 and 1.2 of AmigaDOS and can augment the Metacomco shell. The price for the package is $49.95. Reader Service No. 21.
Metacomco
26 Portland Sq.
Bristol BS2 8RZ
England
44-272-438781

FTL Modula-2 for the Atari ST from **Workman & Associates** is now shipping. The Atari version is fully compatible with existing FTL programs and includes a complete GEM interface. The package sells for $79.95. Reader Service No. 22.
Workman & Associates
1925 East Mountain St.
Pasadena, CA 91104
(818) 791-7979

Miscellaneous

MEMOCOM is now shipping a universal cross-development kit for the Macintosh that includes a table-driven cross-assembler and a MEMULATOR II or MEMULATOR 16 in-circuit EPROM emulator. With the kit, developers can use the Macintosh to assemble and test sources programs for virtually any microprocessor/controller with 24 or less address bits. Both the universal cross-assembler and MEMULATOR II/16 support industry-standard Intel hex, Motorola S-record, and straight binary formats. These output file formats are compatible with most serial EPROM programs. The MAC Universal Cross-Development Kit sells for $725 with a MEMULATOR II and $1,275 with a MEMULATOR 16. The kit is also available for PC-DOS and MS-DOS systems and the Atari ST. Reader Service No. 23.
MEMOCOM
1920 Arbor Creek Dr.
Carrollton, TX 75010
(214) 446-9906

Addison-Wesley has released the Programmer's Online Companion by Steven Capps, a disk-based database reference to *Inside Macintosh*, Volumes 1–4, and the *Apple Numerics Manual*. The companion is a utility program that allows programmers immediate access to bits of information from within any language development system and includes almost all the system calls, system globals, and assembly-language equates found in those volumes. The program controller resides in 5K RAM. The structure assumes a basic knowledge of Pascal or assembly language, *Inside Macintosh*, and the Macintosh itself. The package includes a 3.5-inch disk and 32 pages of documentation. It runs on the Macintosh 128K, Macintosh 512K, Macintosh 512Ke, and Macintosh Plus and sells for $34.95. Reader Service No. 24.
Addison-Wesley Publishing Co.
Rte. 128
Reading, MA 01867
(617) 944-3700

Whitesmiths has announced CXDB, an interactive C source-level cross-debugger for the Motorola MC680x0 line of processors. CXDB is a host-resident debugger available for the VAX and IBM PC that allows embedded 68K programs to be debugged in terms of the original C source code. Features allow users to execute host system commands; print the contents of a C variable function or file with or without assembly-language display; step through single or multiple lines until a specific location is reached, again with or without assembly-language display; create log output for later examination; set and remove breakpoints on a specific C variable, line, or function; disassemble C source lines; obtain in-line help; display stack frames with or without type; and value information for variables in each frame. Prices for CXDB range from $1,800 to $7,000. Reader Service No. 25.
Whitesmiths Ltd.
59 Power Rd.
Westford, MA 01886
(617) 692-7800

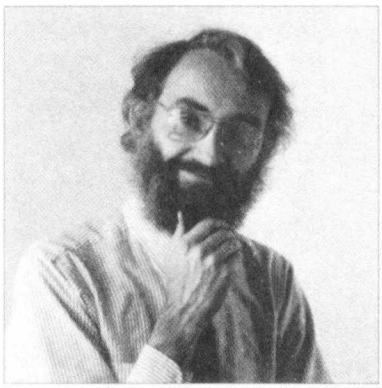
Optotech has introduced Laser DataBank, a "plug-and-play" 5¼-inch write-once read-many (WORM) optical drive subsystem for four popular computer environments: Sun Microsystems' Sun-3; DEC's MicroVAX II; IBM's PC, PC/XT, and 286/386-based computers, including the PS/2 and compatibles; and Apple's Macintosh SE and Macintosh II. The proprietary software also allows data exchange among all four subsystems by writing disks that can be read by any Optotech drive. All the subsystems are built around the Optotech Model 5984 Optical Disk Drive with 400 megabytes of storage per 5¼-inch optical disk drive. Proprietary interface software in each subsystem makes the write-once drive immediately compatible with the host computer's operating system and provides standard file and disk management functions.

The price for the Macintosh Laser DataBank is $3,995. Single-unit prices for the Sun and VAX are $5,950 and $6,950, respectively. The Laser DataBank for PCs is currently available at a single-unit price of $2,995. Reader Service No. 26.
Optotech Inc.
740–770 Wooten Rd.
Colorado Springs, CO 80915-3518
(303) 570-7500

Reduce, from **Northwest Computer Algorithms**, is an interactive software system designed for general mathematical computations of interest to scientists, mathematicians, engineers, and university students. The system has been applied to a variety of problems in many different research areas, including quantum electrodynamics, quantum chromodynamics, electrical network analysis, plasma physics, celestial mechanics, general relativity, numerical analysis, and a variety of engineering problems such as turbine and ship hull design. The code is portable across a wide range of machines. Reduce is now available for the following machines: VAX; Sun; Apollo; Macintosh; and IBM PC, PC/XT, PC/AT, and compatibles. A single-machine license costs $495. Reader Service No. 27.
Northwest Computer Algorithms
P.O. Box 1747
Novato, CA 94948
(415) 897-1302

DDJ

SWAINE'S FLAMES



It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a sub-sub appears to have gone through the long Vaticans and street-stalls of the earth, picking up whatever allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology.

—Herman Melville.

Ahab, standing on the deck of Pequod and searching for the great white whale, overlooked the real monster.

Ben, an editor for another magazine, recently moved to the West Coast. Drawn by the California curls, he went out to walk the waves. Last week the ocean drove his surfboard into his face.

It's been nearly seven years since I moved to the Coast, but I still walk out on the rocks regularly to stand in awe of the big blue beast.

At last fall's Comdex, IBM rolled in with an announcement of the delivery of OS/2 and word that it will be promoting its proprietary Extended Edition of OS/2 over the Standard Edition. The Comdex daily declared the show Very Blue, while several frolicking whales spouted their support.

Among the whales, Lotus Development Corp. announced Agenda. Agenda is the information manager that Jerry Kaplan went to Lotus to develop. It is what he calls an item/category database: the items are short, free-form, textual entities, which the user can assign to points in a user-defined, evolving hierarchy of categories. Agenda could be wildly successful if it sufficiently models the näive view of computers held by many pre-users: that one should be able to type arbitrary information into the machine and retrieve it on demand.

Ashton-Tate's chairman, Ed Esber, had left no doubt as to who would define the dBASE standard when he dared dBASE add-on and compiler vendors to "make his day," which they could do by challenging A-T's claim on the dBASE language (the language itself, not just A-T's implementation). So it came as no surprise to see Marty Winston of Wallsoft at Comdex handing out make-my-day buttons decorated with the international red-circle-and-backslash negation symbol. (It was Winston who brought dBASE aftermarket companies together in a dBASE Standards Committee.) Can a language definition be a product? Esber has not only thrown down the gauntlet to the aftermarket companies but also challenged the industry to deal with this tricky question.

Despite the lukewarm response to IBM's RT, you can find RISC technology implementations for just about any broad-based hardware. Shortly before Comdex, Sun announced its SPARC (Scalable Processor ARChitecture) RISC architecture and by Comdex had licensed the technology to over 40 companies, including Arete. Products based on the Inmos transputer parallel RISC processors are also proliferating. Atari, which incidentally would have won any Most Outrageous Product Name at Comdex competition with its Moses Promiselan, demonstrated a prototype of the Abaq transputer, which can act as a backend for an Atari ST and provide mondo MIPS number crunching and near-photographic graphics. Levco, now a division of Scientific Micro Systems, announced the formation of a TransLink Transputer Developers Group to support developers using its transputer modules in Macs.

Comdex offered a few new implementations of familiar languages. Ryan-McFarland, now a division of Austec, announced an OS/2 version of its ANSI-77 FORTRAN, Lahey offered a PC ANSI-77 FORTRAN for $95, and Prospero had an ANSI-77 FORTRAN for GEM. A number of new Modula-2 implementations or versions were announced, including a new version of the well-regarded Logitech compiler, new FTL versions for several machines, and OXXI's Benchmark Modula-2 for the Commodore Amiga. California Software Products has ported RPG II to PCs. And Borland unloaded a number of language announcements, including Turbo Pascal 4.0.

Meanwhile amid the flotsam and jetsam of the show, I kept sighting ex-editors of ex-programmers' magazines who had jumped ship as their magazines sailed into strange waters. Finally, heading for the pressroom one last time, I ran into Scott Mace, who, having been with *InfoWorld* since before I moved to the Coast, must be the longest-tenured scribe for any personal computer industry publication. A real journalist and a survivor.

And Ben was in the pressroom, the stitches now out, working on his notes.

Michael Swaine

Michael Swaine
editor-in-chief

#136 FEBRUARY 1988

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

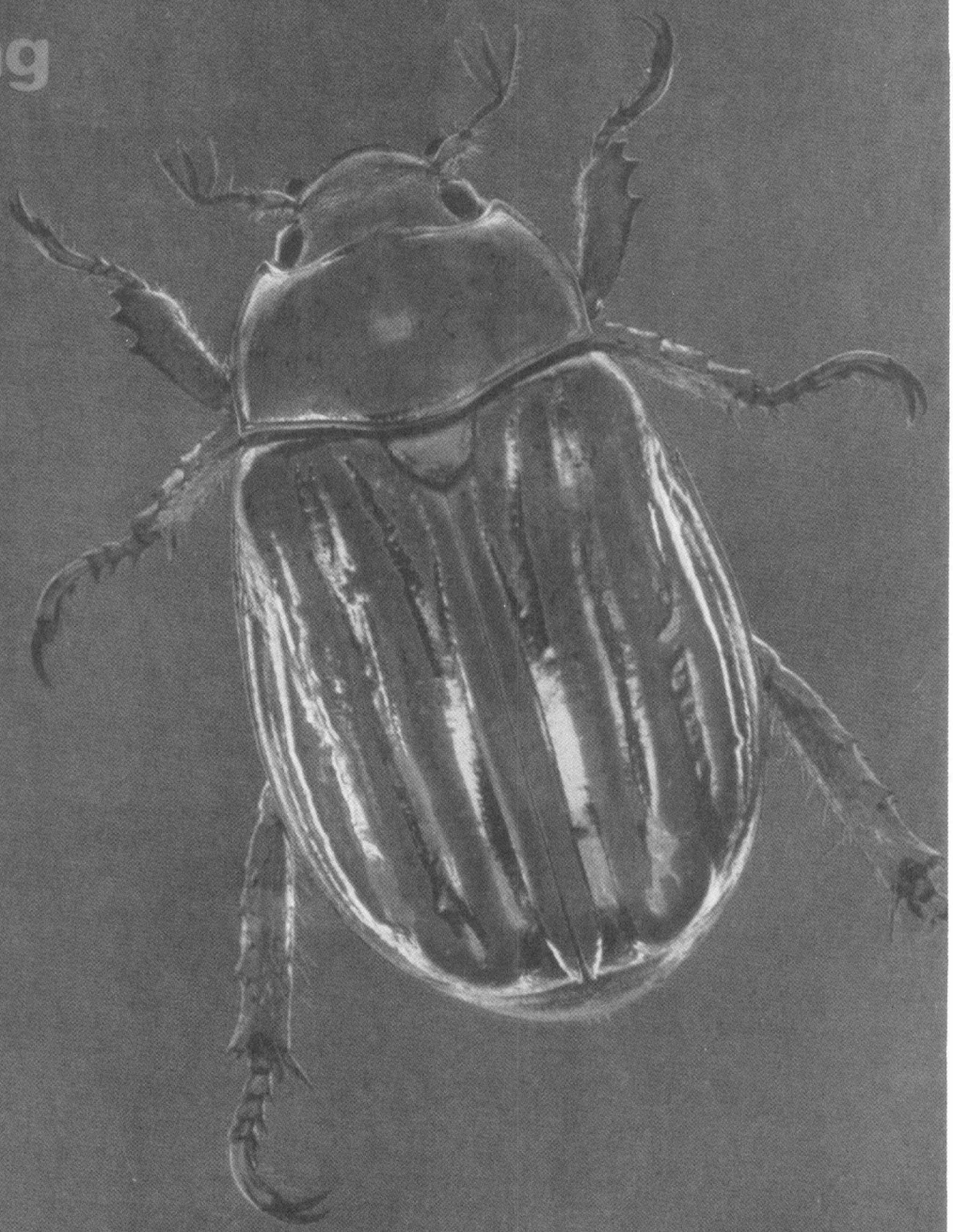
FOR THE PROFESSIONAL PROGRAMMER

Debugging on the 386

Making Serial Links Work

New Product Review Section

Languages: C, Forth, Pascal



0 38351 16562 8 02

ARTICLES

Debugging ▶

Debugging with the 80386 **18**
by Franklin Grossman
The protected mode of the 80386 offers extended software I/O controls which were previously only available through hardware debuggers. Frank shows you how to use the 386's registers and other features to provide advanced debugging support.

A Serial Protocol Analyzer **30**
by Craig Lindley
Craig's SPA offers a slick way to use your PC to visually monitor data flow and handshaking between serial devices.

COLUMNS

C ▶

C CHEST **82**
by Allen Holub
Allen talks about the process of storing and reclaiming program variables using configuration files, followed by a discussion of the merits of awk as a C text query engine.

Pascal ▶

STRUCTURED PROGRAMMING **84**
by Kent Porter
Kent offers some methods for handling DOS critical errors in Turbo Pascal.

Forth ▶

THE FORTH COLUMN **86**
by Martin Tracy
Martin ruminates on the year that was in the Forth estate, and closes with a discussion of how to deal with stream I/O.

REVIEWS

EXAMINING ROOM **116**
coordinated by Ron Copeland
Products examined from the programmer's perspective. This months selections include: The Norton Guides, XO-Shell, MASM 5.0, Soft-Scope V2.55, and CheckMate.

FORUM

EDITORIAL **6**
by Tyler Sperry

RUNNING LIGHT **8**
by Tyler Sperry

ARCHIVES **8**

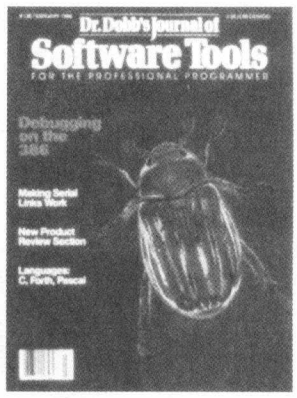
LETTERS **12**
by you

SWAINE'S FLAMES **144**
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX: **129**
Where to go for more information on products mentioned in DDJ.

OF INTEREST **138**
Product news for the programming community



About the Cover

The collective Doctor would like to thank his friends in charge of the Insect Zoo at the San Francisco Zoo for their loan of the Scarab Beetle which appears so prominently on the cover and elsewhere in the magazine. After our art director had bagged a bunch of these beauties, we told him that what we really wanted to show was some fairly sophisticated ways to eliminate the pesky critters. His comments on the inappropriateness of a nuked bug on the cover, and his thorough disgust at the entire concept, forced us settle for this issue's (admittedly tasteful) cover.

Next Issue

DDJ's March issue welcomes back the Mac column from its temporary space crunch hiatus and features some interesting examples of the object-oriented paradigm.

EDITORIAL

The Doctor and Copy Rites

Every so often a letter arrives at the *DDJ* offices that demonstrates a reader's concern over the issue of software copyright. Given that we often publish both articles and source code listings that include an author's "All Rights Reserved" copyright notice, the question has been raised as to what rights (if any) the reader has. A strict reading of copyright law would lead you, after all, to the conclusion that even photocopying a section of a magazine you'd bought would be copyright infringement. Given the current litigious climate of the software industry, it's probably a good idea that we at *DDJ* at least give you a statement of our policies and attitudes concerning copyrights.

Perhaps the most prevalent confusion arises in the infamous Public Domain. The law (in this case) is pretty clear: any work placed in the public domain is available for anyone to use for any purpose. Pretty simple. Yet all too often we receive manuscripts and programs at *DDJ* with copyright notices that read something like this: "Released to the public domain. Commercial use prohibited." A legal paradox. How in the world can you enforce a right that you've given away?

So if you've been wondering why *DDJ* publishes so much code with author copyright notices instead of donations to the public domain, that paradox should be a pretty good clue. Giving away all your rights to a program is an irreversible step and while most programmers like to share their ideas and programs with others, nobody likes to feel exploited. Based on my informal polling of programmers, that's a big problem with giving away software. In the last few years we've seen a number of bulletin boards, information utilities, and bulk disk vendors make more than a little money off the distribution of public domain software.

Given the nature of the industry and *DDJ*'s tradition of sharing information with our fellow programmers, it was clear that relying on articles dealing with PD software alone wasn't going to cut it. The option of standing over our authors with a whip and trying to beat them into submissions (so to speak) just wasn't appealing.

Our solution to the problem is not final or even particularly noble. We just print a magazine every month with articles, columns, and programs. We pay the authors for the rights to print their articles and source code. The authors get the usual fame and glory, a bagful of nickels, and an occasional letter complaining that their program makes the Fastbomb C compiler blow up.

And what about the reader? What are your rights to the software? Well, despite all the lawsuits and paranoia in the software industry, the answer to that is pretty straightforward. All of us—the authors, the editors, and the publisher—agree that once you buy an issue of *DDJ* you are entitled to use that issue's source code listings in any manner you want in the privacy of your own home, for the satisfaction of your intellectual curiosity. This is our family definition of "fair use." Copying the code directly into your own programs and then distributing it is *not* what we call fair use. We call it tacky. Worse, our lawyers call it illegal. If you find yourself in a situation where you'd like to professionally use some code that appears in *DDJ*, please give us or the author a call. Odds are strong that a little politeness will be rewarded with a license that costs little (perhaps just a credit line), and everyone will feel better.

Tyler Sperry
editor

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editor *Rhoda Simmons*
Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Fulfillment Coordinator *Francesca Martin*
Book Marketing Mgr. *Jane Sharninghouse*
Subscription Supervisor *Kathleen Shay*
Newsstand Coordinator *Larry Hupman*
Circulation Assistant *Sarah Frisbie*
Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Marketing/Advertising
Director *Ferris Ferdon*
Marketing Mgr. *Michael Wiener*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 129*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

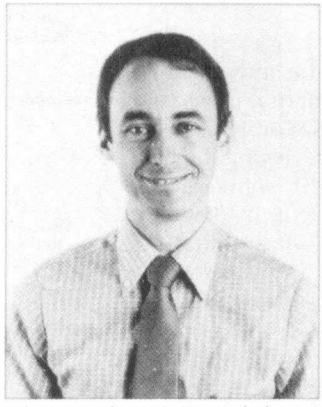
Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $27 per year airmail or $10 per year surface. All other countries add $27 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT



Back in November I asked what I thought was a relatively innocent question: "Are you guys really using those listings or what?" Now, following the deluge of letters that column generated, I have some answers. You use the listings.

The responses were interesting. Some of you were sure I planned to turn *DDJ* into another *Byte* by eliminating all the source code. In fact, several readers threatened to cancel their subscriptions immediately (as they had with *Byte*) if we stopped running the listings. Some readers pointed out that on-line access wasn't a practical option for all parts of the world. And a few, bless you, were charitable enough to note that I was asking for your feedback, not making a threat.

So, to follow up the topic I introduced last November: Yes, there are changes coming for *DDJ*. And no, we're not killing the source listings.

There are some repercussions from printing all that source code, however. As you must have noticed, a long listing can definitely limit the number of articles we can print in a given issue. Programs for the Mac and OS/2 environments seem especially prone to source listings that are much longer than we're used to. We'll do our best to fit things in without resorting to continuing a program through several issues of the magazine, but sometimes there's no completely satisfactory solution. (This issue, for example, we had to postpone Stan Krute's Mac column; next month it will run in its entirety.)

Listings aside, there are some changes planned for the coming months. Chief of these is a new review section coordinated by Ron Copeland. This is not an attempt to woo readers from *PC Magazine*, thank you very much, but an attempt to get more information to you about products of interest *to programmers*. We'll be printing reviews of compilers and libraries, not roundups of printers and surge protectors. Examining Room, a monthly collection of short reviews debuts this month. As with the rest of the magazine, Examining Room is designed around your needs and feedback. If there's a product you're particularly fond of then let Ron or me know by mail, e-mail, or phone.

In other review news, it seems not everyone agreed with Allen Holub's review of C compilers a few months back. Check out the letters section for some of the flames from the CompuServe forum.

On the home front, the chaos around here has abated significantly with the addition of our new managing editor, Monica Berg. Monica is a veteran of innumerable deadline battles, having just completed a tour of duty at *Unix/World* before signing on here at *DDJ*.

One last note on this issue's theme before I go. Robert Ward's *Debugging C* (Que Corp., 1986) isn't a new title, but it is still one of the best books on the subject. He covers the traditional methods and the tricks specific to debugging C as well as the use of sdb and CodeView. Regardless of how you normally work, if you're programming in C, you'll probably learn something from this book. Highly recommended.

Tyler Sperry
editor

ARCHIVES

Ten Years Ago Today

"One principal difficulty with newly evolving computer technology is that software generation tools generally lag corresponding hardware facilities, thus forcing the software engineer to resort to outmoded techniques to produce software systems."—*Gary A. Kildall*, DDJ, *February 1983.*

'This is a long song folks, and tonight its going to be even longer'

"It is often true that along with articles of major import come listings of considerable length. *Dr. Dobb's Journal* traditionally offers useful listings in their entirety, rather than just short excerpts, as illustrations for an article. Sometimes that means we do not have enough space to present the usual variety of articles, because the listings in a particular issue are so voluminous. But we are banking on the fact that the usefulness of the listings will more than compensate for those occasional times when the number of articles is fewer than normal."—*Marlin Ouverson, Editor,* DDJ, *August 1981.*

Yes, but Is It Turbo Compatible?

"Partly as a matter of self preservation, we have become interested in the problem of standards for the Pascal language. The United States Defense Department and many large industrial corporations have recently decided to use Pascal as a base language which they would extend, and possibly alter, to create system implementation languages.... We, and many others in the Pascal User Group, are very much concerned that all this extension and alteration activity will result in Pascal going the way of BASIC for which hundreds of dialects are in common use. We believe that a chance still exists to gain consensus on a substantial family of Pascal extensions for system programming, provided that this can be brought about within the next 6 to 12 months. Unless someone does so before us, we intend to convene a summer workshop for representatives of some of the major using organizations in the hope that such a consensus can be reached."—*Kenneth L. Bowles*, DDJ, *March 1978.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

More Listing Pros and Cons

Dear *DDJ*,

In the November 1987 "Running Light," you asked whether it was sensible to print source code listings. I think it is definitely a good policy for three reasons: some, like me, have no other access to listings; studying the printed listings is helpful in understanding a program; and those who peruse *DDJ* away from a telephone would appreciate printed listings.

You mustn't assume that everyone has ready access to a fast modem or has the money to buy one; I have limited resources and cannot afford a modem. Studying the printed listings is useful in understanding a program and its accompanying article. It would be maddening to read in the text "see line 97" without the listing at hand, especially if you're in a library. Finally, those who flip through *DDJ* in bookstores may decide to buy based on the listings. I have. In fact, I bought several issues at full retail before subscribing. So please don't stop printing source code listings in *DDJ*—they're most useful.

Dan Velting
Grand Rapids, MI 49506

Dear *DDJ*,

November 1987's Running Light provided a long overdue relief. Is it true, you are really going to spare me the monthly trip to my not-so-local newsagent? My sole reason for going there is to pick up a certain computer magazine featuring full-length source listings. If said magazine stops publishing those listings, I in turn shall stay home near the warm oven, which comes in handy with winter almost upon us.

Were I mad enough to use the twisted pair to download the source code absent from *DDJ*'s pages, the post office would charge me $8.40/minute for a phone call to the U.S. If the rare occasion should indeed find me in a spending mood, it would still be in vain, as your Bell standard modem finds itself unable to talk to my V21/V22/V23 DCE.

No, if you want me to keep on getting cold and wet feet, you will continue to print listings in *DDJ*. But who could be that cruel?

Andreas Burmester
2000 Hamburg 70
West Germany

Dear *DDJ*,

Keep the listings! Not all of us have sold our souls to IBM and MS-DOS. Not all of us live where CompuServe is a local call or can afford the charges. Not all of us have a 2,400-bps modem.

The listings are the foundation of *DDJ*. How do you read an article about a program without access to the code? If you do away with the listings, you'll be just another *Byte*—bland pap for computer illiterates.

Mad at you for even suggesting such a thing.

Donald Lashomb
Cranberry Lake, NY

After weeks of unsuccesful debugging, Gordon resorted to a hardware breakpoint.

Turbo Wars

Editor's note: A brief storm of controversy blew up on CompuServe following Allen Holub's comments on C compilers in his October 1987 "C Chest" column. A sample from the thread follows:

Sb: C reviews 10/87
Fm: Jeff Brenton 76703,1065
To: Allen Holub 72407,3564
Dear Allen,

Is Turbo C a better product than Quick C? Probably not, but, until mere mortals can purchase and *receive* Quick C, Turbo is a better product for the "interested in C, but no corporate backing available for my education" user. As I write this note, no one I know of has received Quick C, including people who ordered it directly from Microsoft as part of the version 5.0 upgrade.

On the other hand, Turbo C, though it lacks even a symbolic debugger, has been in my hands since July, and generates better code (for the most part) than v.4.0 of Microsoft C. The fast compile times make it much more "fun" to use than any other compiler I have access to. Its earlier delivery, combined with its performance and low cost, made it possible for Borland to sell more Turbo C packages than Microsoft ever sold of MSC, just in the first months after its release! I wouldn't be surprised if Borland manages to *triple* Microsoft's C compiler sales before Quick C hits the market.

(continued from page 12)

(Borland's currently estimates Turbo C sales at over 150,000.—Ed.)

In spite of learning C in the old separate-everything environment, I still use Turbo C's integrated environment, since I haven't needed to use inline assembly yet. For the sort of quick and dirty stuff I do, I have grown to accept the clunky editor in exchange for the way it drops me into the editor and shows me "the error of my ways." I still use the DeSmet SEE editor for writing and modifying programs, however.

I also appreciate Borland's intention to deliver a "conforming" ANSI C compiler, as opposed to Microsoft's intention to "implement the important aspects" of the proposed ANSI standard.

I will not purchase Quick C by itself; I will wait until I can afford to buy the full Microsoft C v. 5.x package. After all, programming is but a hobby for me at this time, and I don't have a company buying my compilers for me. If Quick C had been available in June, like Turbo C, I *might* have bought it instead, but it wasn't, so I bought my first Borland product ever.

Sincerely,
Jeff Brenton
Woodstock, IL

Dear Jeff,
Your points about availability are well taken. Quick C was supposed to be available by now. As for "conforming" to the ANSI standard (or at least conforming as well as possible to what is, after all, a moving target), both compilers are the same. As for conforming to the de facto Unix standard for the non-ANSI functions, Microsoft is far superior to Turbo. There are problems with the Microsoft Unix support (such as no *ioctl( )*) but I'd rather have no function at all than some function that calls itself *ioctl( )* but has no relationship to the Unix function whatever, as is the case with Borland. *Signal* is another case in point.

—Allen

Dear Allen,
Your example of *ioctl( )* is a rather poor choice. By definition, such a function *must* be OS-specific; the version in Turbo provides "a direct interface to the DOS ioctl call," just as *ioctl( )* provides access to the Unix system call. Sure, it is wonderful to have a function hide the differences, but, since you are already dickering with system-specific code, what good is "portability"?

You call the Draft ANSI Standard for C a moving target, and then point to Microsoft's adoption of the "de facto Unix standard." Which one? Version 7, System III, System V, BSD? If SysV or Berkeley, which release? There are *many* differences, both major and minor, between the C compilers in these Unix systems! The Unix "standard" doesn't *have* to move—it is so broad, that you can call virtually any variation on a function's actions "Unix standard".

That was the whole purpose of X3J11—to establish a true standard, one that allowed some leeway. There are a minimum number of functions, headers and macros that must be there to be "conforming," and their use means portability to other complying compilers. Borland says they are shooting for full compliance. Microsoft says "the important parts." But who decides what is important enough to include? With all the fighting that went on over what was in and out of the standard, ANSI must think *all* of it is important!

I probably will get Quick C and Microsoft C 5.0 eventually. But Turbo C has and will continue to do well for me. Upon its arrival, DeSmet and Eco both left my hard disk. Turbo C takes up less space, generates code better than most, and most of the major bugs have been fixed in my copy.

C you around!

Sb: #Datalight Optimum-C
Fm: Dave Searles 73647,1011
To: Allen Holub 72407,3564
Allen... I just read your review of Datalight Optimum-C in the October *DDJ*. I agree with your assessment of the compiler except for one major point. Optimum-C *does* support debugging as it does produce line number information with the '-g' option. Granted that a Codeview-type debugger isn't included, but I have been very happy with my combination of Optimum-C and Periscope II. I have all the benefits of Microsoft C at less than half the price. And as far as I can tell, I'd put Periscope up against Codeview any day, especially Periscope version III. Nothing beats a *resident* debugger.... Its great when you suddenly notice an anomaly in a previously "debugged" piece of code... just hit the NMI switch and snoop around. Sorry for the sales pitch, but I thought you gave Datalight a cheap shot, even after they have given all of us a cheap shot at a really good compiler.

Fm: Bruce Kitchin 75046,1131
I really take exception to the assumption made by a number of people (usually but not always those who have little or no experience with Turbo C or in one case I remember, a person who used it for a day, ran into a glitch and gave it up and told the world that it was bad) that Borland products are for Sunday hackers. In the *DDJ* article, it was admitted that Turbo C sometimes generated better code than MSC 4.0 but not as good as MSC 5.0. For the price, I hope that MSC 5.0 generates better code, that has not been MSC's strong point up till now.

Without looking at price, I have found it difficult to see much superior about MSC 4.0 over Turbo C and I have found some ways in which Turbo C is superior (not including its greater conformance to the *draft* ANSI proposal which I assume that MSC 5.0 will catch up with, that is just a matter of release dates versus when ANSI updates the draft—a moving target). I've used both compilers with a program that contains many source files with a total line count approaching 20,000. The final code of Turbo C is smaller with both optimizing for space, I've examined a lot of code with both and find a toss up (depends on where you look), and Turbo C gave me better warnings about ambiguous code, about 20 percent of which were latent bugs that I would have had to debug when running.

DDJ

Debugging with the 80386

Notes on real mode debugging with the 386

by Franklin Grossman

Intel's 80386 processor has received a great deal of praise for its improvements over the earlier 80286. *Much* has been written on the excellent speed and (essentially) flat address space of the 386. In developing software, however, there are other concerns that are just as important as operating speed, even though fast development is often a critical factor. In this article I'll discuss some of the advantages of the 386 processor from the perspective of software debugging.

When Intel designed the 80386, it included a new feature: hardware support for debugging. Probably the most important addition of the new debugging support was the inclusion of breakpoint registers, which allow breakpoints on memory reads, writes, and instruction fetches. Although breakpoint registers are powerful, they do have limitations. By using a few other features of the 80386, our company was able to create a software debugger that contains most of the features found in hardware-assisted debuggers. In addition to breakpoint registers, our product, Soft-ICE, uses several protected mode features, such as virtual 8086 mode, paging, I/O privilege level, and breakpoint registers, to add real-time hardware-level breakpoints and other features found only in hardware-assisted debuggers to existing DOS software debuggers. This article describes how these 80386 features work and how our debugger uses them.

Virtual-Machine Capability

All 80x86 real-address-mode software debuggers cause side effects to the program environment. These side effects are caused because the debuggers use memory, interrupt vectors ([*INT 1*], [*INT 2*], and [*INT 3*]) and DOS or BIOS for I/O. Software debuggers are also at risk of being overwritten or affected in other ways by the target program. For this reason, many hardware-assisted debuggers load the debugging code into write-protected memory on an option card. Even this solution can cause side effects because that memory is mapped into the lower Mbyte that is visible to the 80x86 processor in real-address mode, and this address space (such as *C0000H* or *D0000H*) is also used by other adapter cards. By using the 80386 in virtual 8086 mode, though, it is possible to write a debugger that surrounds the DOS environment in a virtual machine without any of the side effects mentioned earlier.

The 80386 provides a virtual 8086 capability intended for use by protected mode operating systems. This feature was necessary because 80386 protected mode is not backward compatible with 8086 executable programs. The 8086 virtual-machine capability is implemented in such a way that a protected mode operating system can allow multiple virtual 8086 tasks that are controlled by the operating system kernel, and so the operating system and other native tasks are isolated from ill-behaved DOS programs. The operating system cannot be overwritten by the DOS program and has complete control over interrupts, I/O, and the memory map.

Several operating systems commercially available today use the virtual 8086 mode of the 80386. These include FlexOS (DRI), PC-MOS/386 (Software Link), VPix (Phoenix Technologies and Interactive Systems), and Windows/

Franklin Grossman is the president of Nu-Mega Technologies and one of the principal designers of the Soft-ICE debugger.

386 (Microsoft).

If you think of the model described earlier but replace the operating system with a debugger, you get some interesting benefits. The debugger can control the 8086 environment without affecting it or being affected by it, and the debugger code does not run in the virtual 8086 task and therefore is not visible to DOS or to DOS programs. To implement a debugger based on this model, the debugger must have many features of an operating system, including a complete I/O system—the debugger cannot rely on DOS or BIOS for I/O.

A protected mode debugger has more control over the virtual 8086 task than is possible with a conventional software debugger. Interrupts are controlled because all interrupts go through the protected mode interrupt table to the protected debugger. (It is the responsibility of the protected debugger to generate the interrupt in the 8086 virtual machine if necessary.) I/O is controlled because the protected debugger has control over which *IN* and *OUT* instructions are passed through to the hardware and which cause exceptions (80386 exceptions are very similar to 8086-style interrupts). The memory map is completely controlled by the 80386 paging mechanism. Using paging, the amount of memory given to the virtual machine can be varied in 4K increments up to 1 Mbyte. In most instances 640K is the right number. Memory pages can also be marked as "not present," causing an exception if a program running in the 80386 virtual machine accesses that memory page and so giving the debugger control over access of memory regions.

This article describes how protected mode features can be used to provide sophisticated debugger breakpoints. The breakpoints are generally implemented by gaining control of the processor when an exception occurs. At this point a debugger window can be popped up or control can be given to a conventional DOS software debugger. This process is described in detail later.

Breakpoint on I/O

A useful feature in debugging device drivers is having breakpoints on *IN* and *OUT* instructions. *IN* and *OUT* instructions execute under control of the protected debugger. The 80386 gives the operating system the ability to trap on accesses to any I/O address, a capability that was included to allow the operating system to "virtualize" the I/O of an ill-behaved DOS program. An example of this would be capturing bytes output to the parallel port in a printer driver application, where the protected mode operating system could send these bytes to a print spooler.

The 80386 allows the protected mode operating system to control the virtual machine's access to I/O ports through a bit mask. The bit mask contains a separate bit for each 8086 I/O port. If the bit is clear, the 80386 lets the *IN* or *OUT* instruction execute normally. If the bit is set, the 80386 generates an exception that is handled by the protected mode operating system. I/O addresses range from 0 to 65,535, so a complete bit mask takes 65,536 bits, or 8K of memory. If the protected mode operating system provides a bit mask of less than 8K, then any accesses to I/O ports that are not covered by the bit mask cause an exception.

Our protected debugger takes the place of the protected mode operating system, using the I/O bit mask to provide breakpoints on *IN* and *OUT* instructions. To set an I/O breakpoint, the debugger sets the bit that corresponds to the specified I/O address. When the target program running in the virtual 8086 task accesses that I/O address, an 80386 exception occurs. An 80386 exception is similar to a software interrupt. At this point the debugger has control, but it does not know if an *IN*, *INS*, *OUT*, or *OUTS* instruction caused the exception.

The protected debugger's exception handler can look at the actual instruction that caused the exception to determine the instruction type. More specific breakpoints are possible by comparing the actual value being output with a predefined value. In the case of an input, the instruction can be single-stepped and the value in the *AL* or *AX* register compared with a predefined value. If these specific criteria are not met, the debugger can give control back to the virtual 8086 task.

One advantage of using the 80386 virtual-machine features to cause I/O breakpoints is that 80386 exceptions occur instantaneously. This may not seem like a revolutionary statement, but the builders of hardware-assisted debuggers and in-circuit emulators have been confronted with this problem for years. As microprocessors become more pipelined, it is difficult to cause an instantaneous breakpoint—for example, most hardware-assisted debuggers generate an NMI (nonmaskable interrupt) when the breakpoint conditions are met. Because the 80386 prefetches and predecodes several instructions before the actual target instruction is executed, the 80386 has actually executed several instructions before it recognizes the NMI.

Breakpoint on Interrupt

When debugging, it is often useful to set a breakpoint on a hardware or software interrupt. You may, for example, want to run a program until it makes a DOS call to read the version number. You could set a breakpoint for *INT 21* with *AH = 30*. Interrupt breakpoints are a natural for protected debuggers.

In our protected debugger, all interrupts go through the protected mode interrupt table that is under complete control of the debugger. It is the responsibility of the protected debugger to get the address from the 8086 virtual mode task's interrupt vector table at *0:0* and transfer control to that address. With a little additional qualification code that compares the interrupt number with a value previously input by the user, you have breakpoint-on-interrupt capability. This method works equally well with hardware or software interrupts.

An End to Hung Programs

Another debugging feature that can be implemented using the virtual machine is the ability to pop your debugger up at any time, even if interrupts are disabled or masked off. Often, when a program is hung, it is

DEBUGGING WITH THE 80386
(continued from page 20)

useful to pop into your debugger and poke around. Conventionally, this is done with an external button that is linked to an option card that causes an NMI. The conventional method often has problems because so many option cards, including most popular multivideo cards, use NMIs.

A protected mode debugger managing a virtual 8086 environment can actually provide this breakout capability through a key sequence. The specific 80386 feature that is used to provide the breakout capability is called privilege levels. To understand privilege levels you must understand a little bit about 80386 protection. The 80386 has four different protection levels, numbered level 0 (highest privilege) through level 3 (lowest privilege). The four different privilege levels can be used for four layers of differing trust levels.

In our debugger application, we need only two levels: level 0 and level 3. The debugger—as you'd expect—runs at level 0, while the virtual 8086 task runs at level 3. A program running at level 0 has complete access to all 80386 protected features. A level 3 program in contrast, cannot access 80386 control registers and other 80386 features. The ability to access I/O ports can be set (by the operating system or in this case by the protected debugger) at any level.

If the privilege level is set to 3, the target program running in the virtual 8086 task has some additional restrictions. Certain instructions that have to do with interrupts cause exceptions. These instructions are *STI*, *CLI*, *LOCK*, *INT*, *PUSHF*, *POPF*, and *IRET*. If you are an assembly-language programmer, you may have noticed that most of these instructions affect the processor interrupt flag. If the 80386 had two interrupt flags—one for the virtual machine and one for the native environment—it would not be necessary to monitor these instructions. Because the 80386 does not keep track of the state of the interrupt flag separately for the virtual machine, the protected debugger must monitor all instructions that cause a change in the state of the interrupt flag.

By getting control when any of these instructions are executed by the target program, it is possible to "virtualize" the interrupt system. In the case of the breakout feature, the only concern is to handle the keyboard interrupt. When the target program disables interrupts, the debugger must continue to get keystroke interrupts. The keyboard interrupt must be handled by the debugger but cannot be passed through to the virtual 8086 task. When keyboard interrupts occur, the protected debugger must monitor the keystrokes looking for a key sequence. If the sequence is found, the debugger is popped up. The tricky part is making sure all keyboard activity meant for the target environment is passed through accurately.

For hard-core systems types, it is worth mentioning that accesses to the interrupt controller mask register must be monitored as well. This is necessary in the cases in which interrupts are masked at the interrupt controller instead of at the processor interrupt flag.

DEBUGGING WITH THE 80386
(continued)

Memory Range Breakpoints

The next debugging feature I'll discuss is memory range breakpoints. Memory range breakpoints are especially useful when an errant program is overwriting a portion of memory. By trapping on the write, you can find the actual code that has gone astray.

To implement memory range breakpoints, the protected debugger uses an 80386 protected mode feature called paging. The paging mechanism in the 80386 was intended for providing demand-page virtual memory in a protected mode operating system. Memory is divided into 4K pages, and the operating system can mark each page as present or not present in the 80386 page tables. If a program is executing and it enters a page that is not present, an exception occurs. It is the responsibility of the operating system paging exception handler to load the actual contents of that page from a mass storage device. Paging takes advantage of the fact that most programs tend to spend most of their execution time in a few concentrated areas.

Again, our debugger takes on the role of an operating system to manage the paging mechanism. When the user specifies a memory range, the debugger must first determine which pages the range covers. The debugger marks these pages not present in the 80386 page tables, and control is given back to the user program. If the user program accesses one of the pages marked not present, an exception occurs, and the debugger's exception handler gets control at this point. The 80386 passes the exception handler the address that was accessed in control register 2 (*CR2*). In most cases the specified memory range does not start exactly on a 4K page boundary. When an exception occurs, the debugger must compare the actual address accessed with the actual range specified. If the access was within the 4K page but not within the specified range, control is given back to the target program.

This boundary condition problem can cause performance of the target program to degrade visibly in some instances, although in most cases the performance hit is negligible. One instance in which the performance drop is noticeable is if the top of the program's stack is in the page but not within the range. Even with this performance hit, range breakpoints using paging are still thousands of times faster than the software simulation technique that some debuggers use.

A refinement of range breakpoints is possible. The 80386 paging mechanism allows pages to be read-protected and write-protected, which gives the debugger the ability to allow memory range breakpoints on read, write, or read/write accesses.

Range breakpoints occur immediately when using the 80386 paging mechanism. The instruction that caused the breakpoint to occur is also restartable, so once the memory is present, the program can continue without missing an instruction. This gives the protected debugger the same advantage over hardware debuggers that it has with I/O breakpoints; breakpoints are not affected by the 80386 instruction pipelining.

Breakpoint Registers

Another 80386 feature that fits in with our protected debugger is the previously mentioned breakpoint registers. Breakpoint registers were designed specifically for debugging purposes. They can be used in real-address mode and can be implemented very easily in a simple terminate-and-stay-resident (TSR) program or directly in a user program.

The 80386 includes four breakpoint registers that can be used to set breakpoints for a byte, word, or double-word. These breakpoints can be on write, read/write, or execute accesses. The four breakpoint registers are named *DR0*, *DR1*, *DR2*, and *DR3*. Each breakpoint register holds the 32-bit linear address of the byte, word, or double-word of interest. The address must be on a word or double-word boundary for those respective data types.

There are two additional registers: *DR6* and *DR7*. *DR6* is a status register that is read after the breakpoint has occurred to determine which of the four breakpoints was triggered. *DR7* is a control register that is written to specify the parameters for a particular debug register—for example, write-only on a double-word. *DR7* also contains two enable bits that must be set to activate the breakpoint.

When a breakpoint goes off, an 80386 exception occurs. The exception handler must read the status register to determine which of the four breakpoints was triggered.

As with the I/O and range breakpoints, you can easily extend the capabilities of the debug registers by adding qualifying code to your exception handler. Other features that our Soft-ICE debugger provides by additional qualification code are breakpoint on read-only and comparison with a data value. A read-only breakpoint can be implemented by decoding the instruction that caused the exception to determine if it is a memory read. If not, control is returned to the target program.

Using breakpoint registers to perform breakpoints on execution has an advantage over the conventional *INT 3* approach. Software debuggers for the 80x86 place an *INT 3* at the address of the desired execute breakpoint. An *INT 3* is used because it is a special single-byte instruction (most interrupts are 2-byte instructions) included in all 80x86 processors specifically for providing breakpoint capability. The *INT 3* approach has the disadvantage that it cannot work in ROM; the breakpoint registers work fine in ROM code.

Triggering Your Favorite Software Debugger

Optimally, the protected debugger should provide all the necessary debugging commands, such as dump, unassemble, modify, single-step, display registers, and so on. Many people, however, are addicted to their favorite language-specific debugger. For these people, it would be nice to extend the capabilities of their existing debugger by adding the features that are possible with the protected debugger. This is possible. The conventional debugger runs in the 8086 virtual machine and the protected debugger runs in protected mode. The DOS environment is still affected by your conventional software debugger, but you can add additional breakpoint capability, such as breakpoint on memory range or I/O ports.

Our Soft-ICE debugger provides both methods. For users who need a complete systems debugger, we provide all the necessary debugging commands. For those who wish to extend the capability of their existing software debuggers, we have a pop-up window that allows them to set sophisticated breakpoints that will trigger their software debugger.

Triggering the conventional software debugger is possible by understanding a little about the way most 80x86 software debuggers work. Most software debuggers use the *INT 3* approach described earlier to provide breakpoints on execution. They also use the 80x86 *INT 1* single-step mechanism. All members of the 8086 line have the capability to single-step the next instruction. The debugger must set a special processor flag called the trap flag, and when the trap flag is set, an *INT 1* occurs after every instruction is executed.

By relying on the *INT 1* vector or the *INT 3* vector pointing to the conventional debugger's breakpoint-handling routines, we can generate *INT1*s or *INT 3*s to wake up a conventional debugger. Most debuggers handle unsolicited *INT* 1s or *INT 3*s beautifully; however, a few will not. The third approach uses NMI. Most debuggers have a method of handling unsolicited breakpoints through the NMI mechanism—for example, CodeView provides this with a command-line switch. They provide this capability to break out of hung programs when the user presses an external button. This button is wired through the PC bus to the NMI pin. By taking advantage of these three conventional breakpoint mechanisms, we can wake up almost any conventional debugger.

A side effect of waking up a debugger unknowingly is a problem with reentrancy. Many debuggers enable interrupts or use DOS for I/O. If you wake the debugger up while the processor is in an interrupt routine, or within MS-DOS or the ROM BIOS, the debugger will fail. Waking up a nonreentrant debugger is still useful for application-level debugging. Many debuggers are mostly reentrant, and you can wake these up at any time; Periscope I and II are examples of these.

Conclusion

To build a complete protected debugger requires several additional components. These include "virtualizing" the video display system to save and restore the screen at any time and likewise the keyboard so users can debug keyboard device drivers. The purpose of this article was to describe debugging features, so I haven't gone into those details. By creatively applying 80386 protected mode operating system features in a real-address mode debugger, you can provide most of the features found in a hardware-assisted debugger with the convenience of a software debugger.

DDJ

Vote for your favorite feature/article.
Circle Reader Service No. 4.

A Serial Protocol Analyzer Program

Debugging the bit stream with Turbo Pascal

by Craig A. Lindley

This article describes the implementation of an RS-232 serial protocol analyzer (SPA) program hosted on an IBM PC computer. It utilizes the multitasking kernel I presented in the July 1987 issue of *DDJ*. The software provided in this article converts a PC into a tool that can be used to monitor most RS-232 connections.

The program was developed as an alternative to spending $5,000–20,000 on a dedicated serial protocol analyzer device. Don't misunderstand, this program (in its present form) doesn't have half the features and functions provided by a dedicated protocol tester, but it performs very well when simply displaying RS-232 data—the function a dedicated analyzer is used for 95 percent of the time. What's even better is that the SPA software is available, so you can tailor it to your specific applications.

This program will find use in many software development labs, computer/data-processing centers, and just about anywhere else that RS-232 devices are used. In this day and age of corporate frugality, buying or borrowing an IBM PC to run an application of this type is probably a lot easier than trying to justify the cost of a dedicated protocol analyzer. Also, you can't run Lotus 1-2-3 or word-processing software on your dedicated protocol analyzer when it isn't being used for its designed function.

Craig A. Lindley, 6 Sutherland Pl., Manitou Springs, CO 80829. Craig has been working with real-time operating systems for many years. He has worked at the Jet Propulsion Laboratory on the implementation of part of the real-time system to be used on the upcoming Galileo spacecraft and is currently employed at Rolm Corp. as a software engineer involved in real-time telephony control.

What Does an SPA Do?

An SPA provides a visual picture of data flowing between two serial devices connected via an RS-232 interface. In addition, it provides a method of monitoring the RS-232 handshake lines (RTS, CTS, DTR, DSR, and so on) in a manner not unlike the two-color LED arrangement used on RS-232 breakout boxes. Figuring out what a serial interface is doing (or is not doing) without a device of this sort is extremely difficult.

The SPA will find use in two major areas—first, in software labs to assist in the development of RS-232 data protocols for new products or devices, and second, in data communication users environments in figuring out why two supposedly compatible serial devices are not talking to each other. I'll give an example of each application.

Consider the development of a terminal emulator program. The SPA could be used to verify that key codes programmed on a special function key were indeed being sent out of the serial port when the special function key was pressed. The SPA could also be used to verify that the terminal program could really generate a break condition on its port lines. Finally, the SPA could be used to verify the terminal emulator's implementation of the Xmodem file transfer protocol.

In a data communications environment, consider the case of a serial printer loosing some of the characters sent to it. The SPA could be used to determine that the printer was sending the XOFF software handshake command but that the source of the serial data was not responding to it by stopping its transmission of data.

These are two of the many possible applications of the SPA program. In a general sense, the SPA can take the guesswork out of serial device interfacing and can be considered a serial interface debugger. Any aids in the debugging process will equate to increased productivity and reduced frustration.

What Is Required

The following is the minimum equipment required for use of the SPA program:

1. An IBM PC, PC/XT, or PC AT (or

compatible) with at least 256K RAM. An AT is required for data rates greater than 4,800 baud.
2. Two serial ports: COM1 configured at address 3F8H using interrupt IRQ4 and COM2 configured at address 2F8H using interrupt IRQ3.
3. The SPA executable program (SPA.COM).

If you would like to experiment with the SPA code, these additional items are required:

1. 256K of additional memory for compiling the program in memory and executing it.
2. Turbo Pascal, Version 3.0 or later, for the IBM PC.
3. The following source code files:
- SPA.PAS (Listing One, page 44)—the main program file
- MENU.PAS (Listing Two, page 66)—the menuing subsystem
- SERIAL.PAS (Listing Three, page 80)—the serial port handlers
- MULTI.PAS—the multitasking kernel presented in the July 1987 issue of *DDJ*, with slight modification.

Personally, I think the SPA is a useful program. In addition, if you examine the software components that make up the SPA, you'll find the following components are useful in themselves:

1. A generalized multitasking kernel written in Turbo Pascal.
2. A generalized 1-2-3-like menuing system that can be converted for use in another application simply by changing its database.
3. A BIOS-independent, interrupt-driven, serial interface package supporting both the COM1 and COM2 ports. This, too, is written in Turbo Pascal.

The moral of this story is, if you can't use the whole program, maybe you can use the pieces.

Connecting the SPA for Use

Two methods exist for connecting the SPA to the RS-232 interface to be monitored. The first I call the passive monitoring connection and is shown in Figure 1, right.

This diagram shows the minimum connections necessary to use the SPA to monitor an RS-232 connection. In this case the SFA does not pass the data through. It merely monitors the data sent between both serial devices.

The additional monitor lines can be connected to any of the serial lines you wish to monitor on the SPAs' screen. The signal connected to: pin 5 will show up as CTS; pin 6 will show up as DSR; pin 8 will show up as CD; pin 22 will show up as RI. With this connection method the PC, which is hosting the SPA program, does not sit between the two ends of the RS-232 cable. The two endpoint serial devices are connected just as they would be without the SPA attached. The SPA is effectively connected in parallel to

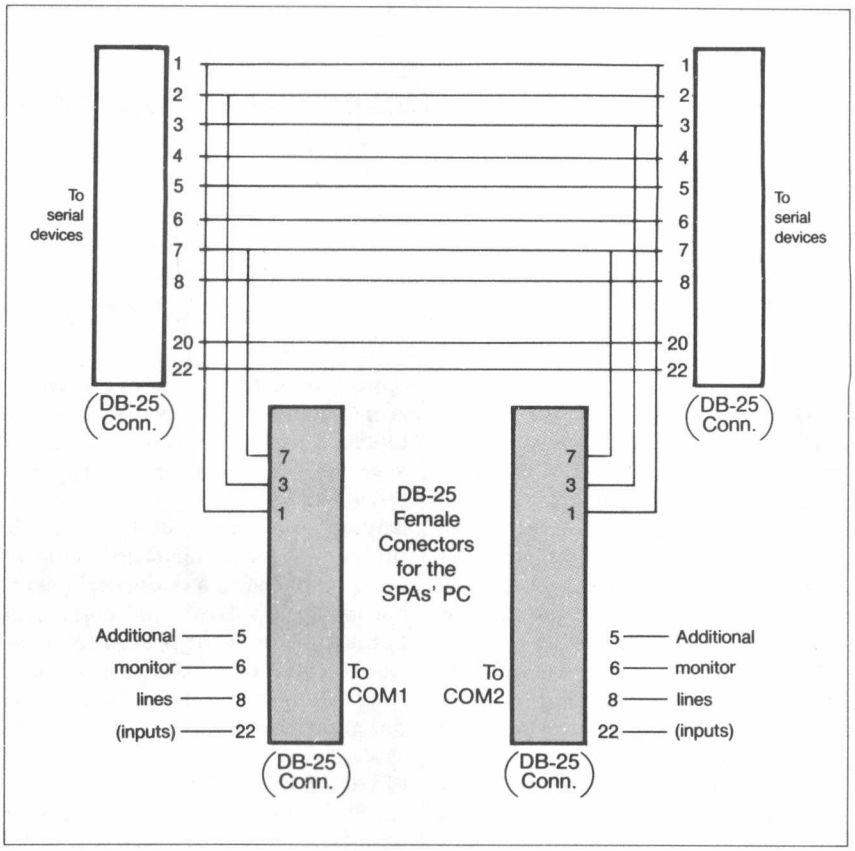


Figure 1: *The passive monitor connection*

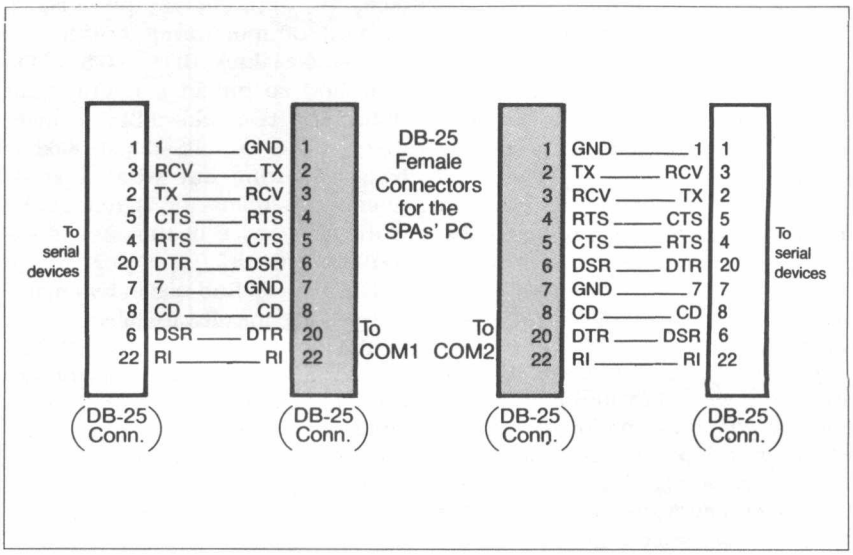


Figure 2: *Pass through monitoring*

the serial data path. The data being transmitted by the devices on both ends of the serial interface is routed to the receive data inputs of COM ports 1 and 2 of the SPA's PC. Also, any of the four available inputs to the COM ports (CTS, DSR, CD, or RI) can be tied to any of the lines of the serial interface being monitored to allow the SPA to display their states visually to the user.

The second connection method I call the pass through connection. Figure 2, page 31, shows the cabling required for this connection method. With this connection, the SPA program sits between both serial devices. All data and handshake line states generated and received by both serial endpoint devices pass through (and can therefore be processed by) the SPA program. This is the mode for which the current SPA software is optimized. It allows the maximum flexibility for future SPA program functionality.

The CD and RI lines may require additional connections if the serial devices being monitored require that these signals are actively driven. The PC serial hardware does not have the driver outputs necessary to drive these signals as it does the RTS and DTS signals. The CD and RI lines from one side could be connected to the same signal pins on the other side.

```
= COM1 ========= Serial Protocol Analyzer  Ver: 1.0 ========== COM2 =
|In Fifo:  0%   OutFifo 0% |       By      |In Fifo:  0% Out Fifo: 0% |
|Stat- BRK:- FE:- PE:- OR:-|Craig Lindley|Stat- BRK:- FE:- PE:- OR:-|
|In - CD:M RI:S DSR:S CTS:S|Display       |In - CD:M RI:S DSR:S CTS:S|
|Out - DTR:S    RTS:S      |Fifo: 3%      |Out - DTR:S    RTS:S      |
=============================================================================
```

Figure 3: *The main SPA display screen*

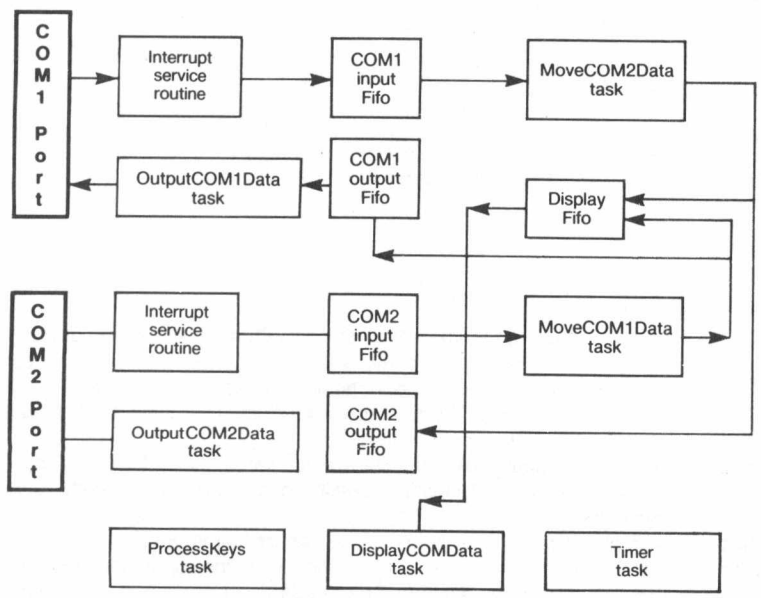


Figure 4: *The block diagram of the SPA program.*

How to Use the SPA

After connecting the SPA's PC to the serial devices using one of the two connection methods, you must execute the SPA program. The SPA program is normally compiled into an executable .COM file called SPA.COM. To execute it just type *SPA* at the DOS prompt.

The first thing you'll notice is that the SPA program verifies the presence of two serial ports (COM1 and COM2 at the proper addresses). Unless two serial ports are found, the program will halt with an error message.

If the hardware verification is successful, the operator is presented with the main display screen, similiar to that shown in Figure 3, page 32. As you can see, this screen presents a lot of information about both the COM1 and the COM2 ports. I'll now describe each item:

In Fifo, *Out Fifo*, and *Display Fifo*: These numbers, expressed in percentages, indicate how full the various FIFOs used by the SPA program are. See the block diagram in Figure 4, page 32, for an explanation of the function of each of these FIFOs.

Note: These percentages are not updated in real time, only every half second. They are shown to provide a feel for how well the SPA is processing the serial data. It is quite possible, however, to have a FIFO overflow occur even though the numbers suggest there is still considerable room in the FIFO.

Stat—BRK, *FE*, *PE*, and *OR*: These are the various status and error conditions of the COM ports in the PC. *BRK* indicates a break condition on the line, *FE* indicates a framing error, *PE* indicates a parity error, and *OR* indicates an overrun error has occurred. See the IBM technical reference manual for an explanation of these conditions. The no-error condition is indicated with a dash whereas the error condition is indicated by an *E* or *B* in the case of a

break.

In—CD, RI, DSR, and *CTS*: These are the various input lines into the COM ports. Their states are continually monitored by the SPA program, and any changes are shown on the display. Their states are either *M* for marking or *S* for spacing.

Out—DTR and *RTS*: These are the two output lines from the COM ports. Their states are also either *M* for marking or *S* for spacing. Note: The SPA program routes what is received on one COM port's DSR line to the other COM port's DTR line, effectivley passing the state through the PC. The same is true of the CTS and RTS lines. This handshake line pass through happens in both directions.

The information line at the bottom of the display informs the user that:

1. If the Esc key is pressed, the menu system will be entered. The highest-level menu is shown in Figure 5, page 34.
2. If the space bar is pressed while data is being displayed, the display will pause until the Enter key is pressed to restart it.
3. If COM1 data is being displayed

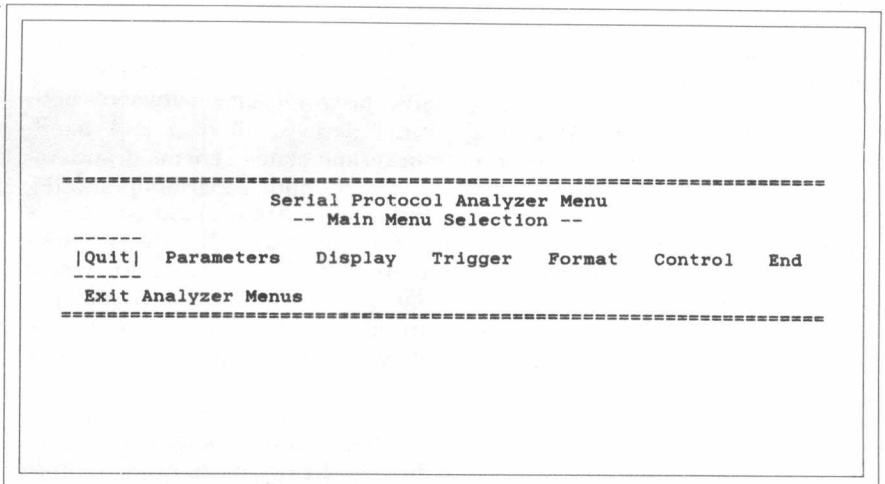


Figure 5: *Main menu of the SPA program*

```
Main Menu - Transition Chars = QPDTFCE
 (0,0,0)
    |
Quit    Parameters   Display   Trigger   Format   Control   End
(1,0,0)  (2,0,0)     (3,0,0)   (4,0,0)   (5,0,0)  (6,0,0)   (7,0,0)
ccode=1 Parameters - Transition Chars = QBSWP
 (2,0,0)
    |
Quit    Baud Rate   Stop Bits   Wrd Length   Parity
(2,1,0)  (2,2,0)     (2,3,0)     (2,4,0)     (2,5,0)
Baud Rate - Transition Chars = Q361249
 (2,2,0)
    |
Quit      300      600     1200     2400     4800     9600
(2,2,1) (2,2,2) (2,2,3) (2,2,4) (2,2,5) (2,2,6) (2,2,7)
        ccode=2 ccode=3 ccode=4 ccode=5 ccode=6 ccode=7
Stop Bits - Transition Chars = Q12
 (2,3,0)
    |
Quit        1        2
(2,3,1)  (2,3,2)  (2,3,3)
         ccode=8  ccode=9
```

Figure 6: *The A tree menu structure (partial)*

menu_entry = RECORD

title —Item or submenu name. This string with a maximum length of ten characters is displayed on the selector line and can be selected by pressing a key corresponding to the first character of its name.

desc —Description of item or submenu up to 40 characters in length. Explanatory text describing the item or submenu.

chars —A string of upper case characters which can be used to select any item on this menu level. This string usually contains the first character of each items title.

index —This entry contains the length of the **"chars"** string above.

ccode —Only non zero at A tree leaf nodes. It is the command code to be processed by the procedure **ProcessCmd** which is associated with this menu entry.

END;

Figure 7: *A Menu Entry Record*

(continued from page 34)

(the data received at the COM1 port), it will be shown in normal video. If COM2 data is being displayed, it will be in reverse video to allow an easy visual distinction.

To set up the SPA program for use, you must enter the menu system using the Esc key. Normally, the setup consists of selecting the proper baud rate, parity, and word length of the serial data stream to be monitored. The defaults chosen for all other setup parameters are adequate for an initial data display. Once the setup is complete, the endpoint serial devices should be enabled to start data transmission. If all is well, the data being passed back and forth between the serial devices should be shown on the SPA's display.

The menu system has the capability of altering the operation of the SPA program. In all, there are 35 possible alterable parameters that determine how the SPA operates. The procedure *ProcessCmd* in the file SPA.PAS (Listing One) shows exactly how each of the menu options is processed. A quick trip through the menuing system will acquaint you with flexibility of the SPA program. Note that data will still be acquired and displayed by the SPA program while you are using the menus. That is the beauty of a multitasking system.

The more important features of the menu system are:

1. Under the Parameters menu, you set the baud rate, parity, and number of stop bits with which the data should be interpreted. The program supports rates from 300 to 9,600; five through eight data bits; odd, even, or no parity; and one or two stop bits. If the serial parameters are set incorrectly, the data displayed on the SPA's screen will be garbage.
2. Under the Display menu, you set whether the data from COM1, COM2, or both is displayed on the screen.
3. Under the Trigger menu, you set up the SPA's triggering capability. Here you input the channel (COM1 or COM2) to trigger from, the single-byte trigger data pattern, and the trigger mode (display data until trigger or display data after trigger). You can also stop (wait for) the trigger in this submenu. Note: After the trigger parameters are set, the trigger must be enabled before it goes into effect.
4. The Format submenu is where the format of the displayed data is set. The options are:
   - ASCII data without handshake (default)
   - ASCII data with handshake (handshake shown in hex)
   - Hex data without handshake
   - Hex data with handshake

   When data is displayed with the handshake option, the port input lines that were acquired when the data was acquired and certain COM port error conditions will be displayed. The information is bit encoded into the displayed hex byte as shown in Table 1, page 37.

   The final option in the Format menu is simply called SPACE. This is a toggle that determines whether the data displayed on the screen is separated or not—in other words, whether a space will be inserted between each displayed data item. If a space is being inserted and this menu option is selected, the spacing will be stopped; the reverse is also true.
5. The Control submenu contains several commands that control the operation of the SPA program. Data acquisition can be stopped and started; the display can be cleared; and finally, the SPA program can be reset to its power-on defaults.
6. The Quit menu option returns you to the main SPA display screen, and the End option ends the operation of the SPA program completely and returns control to DOS. Note: Selecting Quit from any submenu will always bring you back up one menu level.

The complete hierarchical menu structure is too big to list. You can look at the procedure *Init_Menu* in the file MENU.PAS for more information. Also, a short textual message is displayed with each possible menu selection. This is a limited form of context-sensitive help to guide you through the menuing system without your having to consult any documentation.

The SPA Program's Architecture

Now that you understand how to use the SPA program, I'll spend a few minutes discussing the underlying technical aspects of the program's operation. For the following discussion, please refer to Figure 4.

As shown in this block diagram, the SPA program is made up of seven relatively independent tasks, most of which are bound to a FIFO. In summary, the tasks perform the following:

The tasks *MoveCOM1Data* and *MoveCOM2Data* perform the same function on different FIFOs for different COM ports. These tasks perform three basic functions. First, they move serial data and handshake information from their respective input FIFO to the opposite output FIFO. This makes the serial data path through the PC. Second, the serial data is tagged with a source identifier (either from COM1 or COM2), and if the display data flag is set, the data is moved into the display FIFO. Finally, code in these tasks provides the data triggering function.

The tasks *OutputCOM1Data* and *OutputCOM2Data* again are identical in function but access different FIFOs. They check to see whether there is data in their respective output FIFOs, and if so they send it out the COM port.

Bit Numbers							
7	6	5	4	3	2	1	0
CD	RI	DSR	CTS	BRK	FE	PE	OR

Table 1: *Correspondence between data bit encoded and displayed hex byte*

The task *DisplayCOMData* removes entries from the display FIFO, formats them according to the SPA *user's specification*, and displays them on the SPA's display.

The *Timer* task runs every half second. Its purpose is to update the information on the main SPA display screen. Specifically, it updates the handshake information and the buffer usage information displayed to the user.

The final task, *ProcessKeysTask*, monitors the SPA's keyboard and processes all user keystrokes. The menuing system is invoked via this task.

The Menuing System

As previously mentioned, a hierarchical series of menus is used to control the operation of the SPA program. The menuing system is modeled after Lotus 1-2-3 because of its ease of use and the appropriateness of its operation. A menu item or submenu is selected by using the cursor arrow keys to place the reverse-video selector box over the desired item and pressing Enter or by typing the first character of the item's name. The user of the SPA program can move through almost the entire menuing sustem by pressing only single keys. In only one instance—the selection of a trigger byte—is the use of more than one key necessary.

Five procedures from the file MENU.PAS (*DisplayMenu*, *ProcessCr*, *ProcessMenu*, *ProcessCmd*, and *DoMenu*) are used in conjunction with a large data structure to provide the Lotus 1-2-3-like menuing system. These procedures implement an A-tree walk through the menu data structure. For more information on the A (or Awkward) tree structure, see the article "Indexing Open Ended Tree Structures" by John Snyder in the May 1984 issue of *Byte* magazine.

The A-tree menu data structure is rather inefficiently, but simply, realized in the SPA program using a three-dimensional array of menu entry records. A menu entry record is used for each item in the entire menu. Figure 6, page 34, shows a portion of the menu structure. Approximately 24K of data is required to support the menuing system as the array of menu entry records is very sparse.

The numbers in ( ) are the array indices of the menu entry record corresponding to this menu item or submenu. All items with an assigned ccode are leaf or terminal nodes of the A tree. These codes are processed by the procedure *ProcessCdm*. Transition characters are those which cause movement to a different menu level. Typically, they are the first character of each menu item. The complete A tree menu structure for the SPA program is built by the procedure Init_Menu. Figure 7, page 34, shows a detailed breakdown of the menu entry record used in the A-tree structure.

The following keys have special significance while using the menus. They are:

Enter—Selects the currently highlighted menu item. If the item is a submenu, the submenu is displayed with its own selectable options. If the item is a leaf node of the tree, it returns a unique command code (*ccode*) specifying an action to be performed.

Cursor arrow keys—Move the highlighted, reverse-video selector box through the items in a menu. Full selection wrapping is supported.

Esc—Terminates the menuing sys-

tem and returns to the main SPA display screen.
Home—Moves the selector to the first menu entry.
End—Moves the selector to the last menu entry.

Hopefully, the operation of the menuing system will be discernable from the commented listings. Possibly a future article could be written to discuss the menus in detail if interest warrants it.

Changes to the Multitasking Kernel

I've made two changes to the kernel presented in the July issue of *DDJ* to augment its capabilities for use with the SPA program. They are:

1. The constant *task_stack_size* has been increased to 1,000 bytes. This was necessary because of procedure nesting depths, the use of interrupts, and Turbo Pascal's use of BIOS calls. Each task is assigned a stack size of 1,000 when forked. The seven tasks utilized for the SPA program consume approximately 7K of memory for their stacks.
2. Substantial portions of the procedures *Yield* and *Wait* have been recoded into assembly language. This minimizes the task-switching overhead experienced by the CPU. Listing Four, page 80, shows the assembly-language recoding. This change was not absolutely necessary for the operation of the SPA, but it seemed to help the performance of the data display when operating above 2,400 baud. By way of comparison, the hand-optimized assembly-language code uses half the instructions generated by the Turbo Pascal compiler for the equivalent code.

Possible Enhancements

As presented, the SPA program is a rather basic tool. Its present form is in part attributable to the application for which I have been using it. Other applications will require different incarnations of the basic program. Additional features and functions will begin to suggest themselves with prolonged use of the SPA. Already my wish list is growing. Example excerpts are:

1. A setup file that would be read at the start-up of the SPA program that would configure the SPA as I had previously left it. Included in this setup could be serial parameters, trigger info, and display-formatting information.
2. Logging of captured data to a file or printer. Currently, the Shift-PrtScn key sequence must be used to get hard copy of the displayed data.
3. The output of canned serial data to a COM port on the occurrence of a trigger.
4. Increasing the trigger capability from a single byte to multibyte patterns or strings.
5. Other serial data formats—for example, BiSync, HDLC, and SDLC.

Maybe inspired readers can add these functions (and more) and graciously provide me with the code modifications. If I receive enough suggestions and modifications, I'll collect them into a usable form and write another article describing them.

Modification and Testing Issues

At the present time, the SPA program has never been tested (for any length of time) above 2,400 baud because I lack the equipment to test it throughly. It has been tested substantially at 1,200 baud in the development of an Xmodem protocol. If you require baud rates above 2,400 for your application, you may need to make minor changes to the program to allow it to keep up. With rates above 4,800 baud, I recommend using an IBM PC AT. In the future I hope to test it out thoroughly above 2,400 baud (read, as soon as my project at work slows down some).

As mentioned previously, the software as presented is optimized for the pass through mode of connection. If you would rather use the passive monitoring connection, you can modify the software to improve its performance. The modifications are as follows:

1. Remove the section of code in SPA.PAS (in main) that forks off the *OutputCOM1Data* and *OutputCOM2-*

Data tasks. These aren't needed for the passive monitor conection because the SPA's PC doesn't have to output any data.
2. Modify the code in the tasks *MoveCOM1Data* and *MoveCOM2Data* so that they don't move the serial data to the output FIFOs. From *MoveCOM1Data*, remove the line:

PutSerialData(Sd,COM2_Output_Fifo);

and from *MoveCOM2Data*, remove the line:

PutSerialData(Sd,COM1_Output_Fifo);

These changes will eliminate two unnecessary tasks from being executed and will speed up (slightly) the processing of data in two other tasks. The original code is required, however, when the SPA is configured for the pass through method of connection.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 *to Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 44.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Listing One *(Text begins on page 30.)*

```
{***********************************************}
{***                                         ***}
{***                Turbo Pascal             ***}
{***         Serial Protocol Analyzer        ***}
{***                written by               ***}
{***              Craig A. Lindley           ***}
{***                                         ***}
{***    Ver: 2.0     Last update: 08/15/87   ***}
{***                                         ***}
{***********************************************}

{$K-,U-,C-,G30,D-}

{ ------- Notes on compiler directives --------- }
{ K-     No stack checking otherwise multitasking}
{         kernal will not run.                   }
{ U-,C-  Turn off user break checks to speed     }
{         screen I/O.                            }
{ G30,D- Buffer standard input device (keyboard) }
{         and disable device checks. This makes  }
{         keyboard respond much faster and be    }
{         buffered.                              }

CONST

 HexDigits:  STRING[16] = '0123456789ABCDEF';

 AsciiStrs: ARRAY[0..31] OF STRING[3] =
 ('Nul','Soh','Stx','Etx','Eot','Enq','Ack','Bel',
  'Bs' ,'Ht' ,'Lf' ,'Vt' ,'Ff' ,'Cr' ,'So' ,'Si' ,
  'Dle','Dc1','Dc2','Dc3','Dc4','Nak','Syn','Etb',
  'Can','Em' ,'Sub','Esc','Fs' ,'Gs' ,'Rs' ,'Vs');


 SerialDataFifoSize = 2000;  {serial data fifo size}
 DisplayFifoSize    = 3000;  {display fifo size}

 {$I multi.pas}              {include the}
                             {multitasking kernel}

TYPE

 FullString  = STRING[255];
 Str80       = STRING[80];
 Str3        = STRING[3];

 DataRec = RECORD
  Data,
  Status:  Byte;
 END;

 DisplayRec = RECORD
  Tag: (FromCOM1,FromCOM2);
  DR: DataRec;
 END;

{This fifo overhead structure is the same for}
{all fifo types regardless of the items to be}
{stored in the fifo. Two types are fifos are }
{defined.}

 OverHead = RECORD      {fifo overhead data}
                        {structure}
    Count,              {# of items in fifo}
    Inptr,              {ptr to where items are}
                        {stored}
    Outptr:   Integer; {ptr to where items are}
                        {fetched}
    NotEmpty,          {ptrs to waiting tasks}
    NotFull:  tcbptr;
 END;


 {definition of a serial data fifo}
 SerialDataFifo = RECORD
   Ovd:  OverHead;           {fifo overhead}
   Data: ARRAY[1..SerialDataFifoSize]
           OF DataRec;       {fifo data area}
 END;

 {definition of display fifo}
 DisplayFifoType = RECORD
   Ovd:  OverHead;
   Data: ARRAY[1..DisplayFifoSize]
           OF DisplayRec;
 END;

 DisplayTriggerType = (Before,After);

VAR

 regs:        register_type;

 {storage for the original IRQ3 & 4 code segment}
 {and instruction pointer addresses}

 OldIRQ3_CS,
 OldIRQ3_IP,
 OldIRQ4_CS,
 OldIRQ4_IP:  Integer;

 {UART status storage variables}

 OldCOM1_Status,
 OldCOM2_Status,
 COM1_Status,
 COM2_Status: Byte;

 {Display formatting boolean flags}

 UpdateScreenStatus,
 AsciiDisplay,
 HandShakeDisplay,
 AddSpace,

 {Data display boolean flags}
 {If true the data from the specified COM port}
 {is tagged and then moved into the display fifo}

 COM1_Display_Data,
 COM2_Display_Data,

 {Data Acquisition boolean flags}
 {Controls acquisiton of data by the Interrupt}
 {Serivce Routines. If true then serial data is}
 {stored by the ISR.}

 COM1_Data_Acquire,
 COM2_Data_Acquire,

 {Variables used for the triggering function}

 TriggerEnabled,
 COM1_Is_Triggered,
 COM2_Is_Triggered:  Boolean;
 TriggerPattern:     Integer;
 TriggerMode:        DisplayTriggerType;

 {Fifo declarations}

 COM1_Input_Fifo,
 COM2_Input_Fifo,
 COM1_Output_Fifo,
 COM2_Output_Fifo:   SerialDataFifo;

 DisplayFifo:        DisplayFifoType;

 {Screen formatting strings built at run time to}
 {format the screen.}

 Line1, Line2,
 Line3, Line4,
 Line5, Line6,
 Line24:        Str80;

 {Cursor storage for DisplayCOMData procedure}

 OldXPos,
 OldYPos:  Integer;

 {Lock for screen control}
 ScreenAccess: Semaphore;

{************ Begin FIFO Procedures ************}

PROCEDURE Init_Fifos;

 PROCEDURE Initialize_fifo(VAR o:OverHead);

 {Initialize a fifo's overhead data structure.}
 {This procedure will work with any type fifo.}
 {This makes the fifo appear empty.}

 BEGIN

    o.Count  := 0;        {count is empty}
    o.Inptr  := 1;        {ptrs to 1st entry}
    o.Outptr := 1;        {put in and take out at}
                          {entry 1}
    o.NotEmpty :=NIL;     {signals to nil}
    o.NotFull  :=NIL;
```

(continued on page 46)

Listing One *(Listing continued, text begins on page 30.)*

```
 END;

BEGIN

   Initialize_fifo(COM1_Input_fifo.Ovd);
   Initialize_fifo(COM1_Output_fifo.Ovd);
   Initialize_fifo(COM2_Input_fifo.Ovd);
   Initialize_fifo(COM2_Output_fifo.Ovd);
   Initialize_fifo(DisplayFifo.Ovd);

END;


PROCEDURE PutSerialData (d:DataRec;
                         VAR f:SerialDataFifo);

BEGIN

   WITH f.Ovd DO
   BEGIN                {check if fifo full}
      IF Count = SerialDataFifoSize THEN
      BEGIN             {if so go to sleep}
         waitfor := addr (NotFull);
         wait;
      END;              {when not full add}
      Count:=Count+1;   {one more to count}
      f.data[Inptr]:=d; {store the data record}
      Inptr:=Inptr+1;   {bump input pointer}
      IF Inptr > SerialDataFifoSize THEN
         Inptr:=1;      {wrap ptr if necessary}

      {if waiters for this fifo wake them}

      IF NotEmpty <> NIL THEN
         send(NotEmpty);
   END;

END;


PROCEDURE GetSerialData (VAR f:SerialDataFifo;
                         VAR d:DataRec);

BEGIN

   WITH f.Ovd DO
   BEGIN               {check if fifo empty}
      IF Count = 0 THEN
      BEGIN            {if so go to sleep}
         waitfor := addr (NotEmpty);
         wait;
      END;
                       {when data is available}
      Count:=Count-1;  {one less to count}
      d :=f.data[Outptr];  {get the data record}
      Outptr:=Outptr+1;{bump output pointer}
      IF Outptr > SerialDataFifoSize THEN
         Outptr:=1;    {wrap ptr if necessary}
      {if waiters for this fifo wake them}

      IF NotFull <> NIL THEN
         send(NotFull);
   END;

END;


PROCEDURE PutDisplayData (d:DisplayRec;
                      VAR f:DisplayFifoType);

BEGIN

   WITH f.Ovd DO
   BEGIN                {check if fifo full}
      IF Count = DisplayFifoSize THEN
      BEGIN             {if so go to sleep}
         waitfor := addr (NotFull);
         wait;
      END;              {when not full add}
      Count:=Count+1;   {one more to count}
      f.data[Inptr]:=d; {store the data record}
      Inptr:=Inptr+1;   {bump input pointer}
      IF Inptr > DisplayFifoSize THEN
         Inptr:=1;      {wrap ptr if necessary}

      {if waiters for this fifo wake them}

      IF NotEmpty <> NIL THEN
         send(NotEmpty);
   END;

END;


PROCEDURE GetDisplayData (VAR f:DisplayFifoType;
                          VAR d:DisplayRec);

BEGIN

   WITH f.Ovd DO
   BEGIN               {check if fifo empty}
      IF Count = 0 THEN
      BEGIN            {if so go to sleep}
         waitfor := addr (NotEmpty);
         wait;
      END;
                       {when data is available}
      Count:=Count-1;  {one less to count}
      d :=f.data[Outptr];  {get the data record}
      Outptr:=Outptr+1;{bump output pointer}
      IF Outptr > DisplayFifoSize THEN
         Outptr:=1;    {wrap ptr if necessary}

      {if waiters for this fifo wake them}

      IF NotFull <> NIL THEN
         send(NotFull);

   END;

END;

{Include the menuing system}

{$I menu.pas}

{Include the serial procedures}

{$I serial.pas}

{********* Additional Serial Procedures *********}


PROCEDURE SetBreak (PortAddr:Integer; State:Boolean);

{Controls the break generation for the specified}
{COM port.}

VAR

 Temp:  Byte;

BEGIN

   {Read the LineControl reg of the 8250}
   {either set or reset the break bit D6}
   {as specified. Write the new reg value}
   {back to the port}

   Temp := port[PortAddr+LineControl];
   IF State THEN
      Temp := Temp OR $40
   ELSE
      Temp := Temp AND $BF;
   port[PortAddr+LineControl] := Temp;

END;


PROCEDURE MakeHandShake (PortAddr:Integer;
                         Status:Byte);

VAR

 Temp:   Byte;

BEGIN

   {Set the bits in the ModemControl reg of the}
   {specified COM port according to the bits}
   {of the variable Status. This procedure is}
   {used to always force the handshake lines of}
   {the COM ports to agree}

   Temp := port[PortAddr+ModemControl];

   IF (Status AND BrkBit) <> 0 THEN
      SetBreak(PortAddr,True)
   ELSE
      SetBreak(PortAddr,False);
```

(continued on page 48)

Listing One *(Listing continued, text begins on page 30.)*

```
    IF (Status AND CTSBit) <> 0 THEN
       Temp := Temp OR $02
    ELSE
       Temp := Temp AND $FD;

    IF (Status AND DSRBit) <> 0 THEN
       Temp := Temp OR $01
    ELSE
       Temp := Temp AND $FE;

    {Store the new value of the ModemControl}
    {register back}

    port[PortAddr+ModemControl] := Temp;

END;


{************** Display Procedures *************}

PROCEDURE BuildDisplay;

{Setup the main SPA display screen}
{The OldCOM?_Status variables are consciencely}
{clobbered to force the screen to be updated}
{after it is built or rebuilt after the menus}
{are displayed}

BEGIN

   OldCOM1_Status    := $FF;
   OldCOM2_Status    := $FF;

   WriteStringAt(Line1,High,1,1);
   WriteStringAt(Line2,Low,1,2);
   WriteStringAt(Line3,Low,1,3);
   WriteStringAt(Line4,Low,1,4);
   WriteStringAt(Line5,Low,1,5);
   WriteStringAt(Line6,Low,1,6);
   WriteStringAt(Line24,Rev,1,24);

END;

PROCEDURE DisplayHandShakeStatus (PortAddr:Integer;
                                  InStatus,
                                  OutStatus:Byte);

CONST

 StatLineNum = 3; {screen line of line status}
 InLineNum   = 4; {screen line of in handshake}
                  {lines}
 OutLineNum  = 5; {screen line of out handshake}
                  {lines}

 CDOffset  = 11;  {offsets on a screen line for}
 RIOffset  = 16;  {the individual items to be}
 DSROffset = 22;  {displayed}
 CTSOffset = 28;
 BRKOffset = 13;
 FEOffset  = 18;
 PEOffset  = 23;
 OROffset  = 28;
 DTROffset = 12;
 RTSOffset = 22;

VAR

 DisplayOffset:  Integer;
 Ind:            Char;

BEGIN

   {Where an item is displayed is determined in}
   {part by which COM status is being displayed}
   {COM1's offset is 0 whereas COM2's offset is}
   {51 character positions.}

   IF PortAddr = COM1 THEN
      DisplayOffset := 0
   ELSE
      DisplayOffset := 51;

   IF (InStatus AND BrkBit) <> 0 THEN
      Ind := 'B'
   ELSE
      Ind := '-';
   WriteStringAt(Ind,High,
                 BRKOffset+DisplayOffset,
                 StatLineNum);

  IF (InStatus AND FEBit) <> 0 THEN
     Ind := 'E'
  ELSE
     Ind := '-';
  WriteStringAt(Ind,High,
                FEOffset+DisplayOffset,
                StatLineNum);

   IF (InStatus AND PEBit) <> 0 THEN
      Ind := 'E'
   ELSE
      Ind := '-';
   WriteStringAt(Ind,High,
                 PEOffset+DisplayOffset,
                 StatLineNum);

   IF (InStatus AND ORBit) <> 0 THEN
      Ind := 'E'
   ELSE
      Ind := '-';
   WriteStringAt(Ind,High,
                 OROffset+DisplayOffset,
                 StatLineNum);

   IF (InStatus AND CDBit) <> 0 THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 CDOffset+DisplayOffset,
                 InLineNum);

   IF (InStatus AND RIBit) <> 0 THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 RIOffset+DisplayOffset,
                 InLineNum);

   IF (InStatus AND DSRBit) <> 0 THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 DSROffset+DisplayOffset,
                 InLineNum);

   IF (InStatus AND CTSBit) <> 0  THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 CTSOffset+DisplayOffset,
                 InLineNum);

   {Note: The OTHER COM port is consulted about}
   {the output status. DTR of this port should}
   {equal DSR of other port. RTS of this port}
   {should equal CTS of other port}

   IF (OutStatus AND DSRBit) <> 0 THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 DTROffset+DisplayOffset,
                 OutLineNum);

   IF (OutStatus AND CTSBit) <> 0  THEN
      Ind := 'M'
   ELSE
      Ind := 'S';
   WriteStringAt(Ind,High,
                 RTSOffset+DisplayOffset,
                 OutLineNum);

END;


PROCEDURE DisplayBufferStatus;

VAR

 PerCentage:  Integer;
 PerCentStr:  Str80;

BEGIN

 PerCentage := Round((COM1_Input_Fifo.Ovd.Count/
                      SerialDataFifoSize) * 100);
 Str(PerCentage:2,PerCentStr);
 WriteStringAt(PerCentStr + '%',High,12,2);

 PerCentage := Round((COM1_Output_Fifo.Ovd.Count/
                      SerialDataFifoSize) * 100);
 Str(PerCentage:2,PerCentStr);
 WriteStringAt(PerCentStr + '%',High,26,2);

 PerCentage := Round((COM2_Input_Fifo.Ovd.Count/
                      SerialDataFifoSize) * 100);
 Str(PerCentage:2,PerCentStr);
 WriteStringAt(PerCentStr + '%',High,63,2);

 PerCentage := Round((COM2_Output_Fifo.Ovd.Count/
                      SerialDataFifoSize) * 100);
```

```
   Str(PerCentage:2,PerCentStr);
   WriteStringAt(PerCentStr + '%',High,77,2);

   PerCentage := Round((DisplayFifo.Ovd.Count/
                        DisplayFifoSize) * 100);
   Str(PerCentage:2,PerCentStr);
   WriteStringAt(PerCentStr + '%',High,47,5);

END;


FUNCTION FormatCharAscii (Num:Integer) : Str3;

BEGIN

   IF Num < ORD(' ') THEN
       FormatCharAscii := AsciiStrs[Num]
   ELSE
       FormatCharAscii := chr(Num);

END;

FUNCTION FormatCharHex (Num:Integer) : Str3;

BEGIN

   FormatCharHex :=
      HexDigits[(Num SHR 4) AND $000F +1 ] +
      HexDigits[(Num AND $000F) + 1];

END;


FUNCTION DisplayData (D:DisplayRec) : Integer;

VAR

 VideoAttrib:  AttribType;
 Temp:         STRING[7];

BEGIN

   WITH D DO
   BEGIN
      {If data from either channel should be}
      {displayed then...}

      IF (((Tag = FromCOM1) AND COM1_Display_Data)  OR
          ((Tag = FromCOM2) AND COM2_Display_Data)) THEN
      BEGIN
         {set video attribute depending upon}
         {which COM channel it is from}
         IF Tag = FromCOM1 THEN
            VideoAttrib := Low
         ELSE
            VideoAttrib := Rev;

         {choose ASCII or Hex format}
         IF AsciiDisplay THEN
            Temp := FormatCharAscii(DR.Data)
         ELSE
            Temp := FormatCharHex(DR.Data);

         IF HandShakeDisplay THEN
            Temp := Temp + ':$'+
               FormatCharHex(DR.Status);

         {write serial data string to display}
         WriteString(Temp,VideoAttrib);

         {if formatting with spaces}
         IF AddSpace THEN
         BEGIN
            {output space to display and add}
            {a space to string for length calc}
            write(' ');
            Temp := Temp + ' ';
         END;

         {ret length of formatted item}
         DisplayData := length(Temp);
      END
      ELSE
         {if no data ret 0 length}
         DisplayData := 0;
   END;

END;

{*********** Miscellaneous Procedures ***********}

PROCEDURE Init_Program;

{Perform default initialization for protocol}
{analyzer program}

BEGIN

   Init_Fifos;

   COM1_Data_Acquire := True;
   COM2_Data_Acquire := True;

   COM1_Display_Data := True;
   COM2_Display_Data := True;

   TriggerEnabled    := False;
   COM1_Is_Triggered := False;
   COM2_Is_Triggered := False;

   AsciiDisplay      := True;
   HandShakeDisplay  := False;
   AddSpace          := True;

   COM1_Status       := 0;
   COM2_Status       := 0;

   {set serial com defaults 1200 baud, 8 bit word,
   {1 stopbit, no parity}
   {both COM ports always set the same}

   COM_Rate     := 8;          {1200 baud}
   COM_StopBits := 1;
   COM_DataBits := 8;
   COM_Parity   := none;

   SetNewCOMParameter; {apply the parameters}

END;


PROCEDURE VerifyHardware;

CONST

 TestByte = $5A;   {try to store and retrieve}
                   {this value}
 SafeByte = $03;   {set default 8 bits 1 stop}

BEGIN

   ClrScr;
   {just in case of reentry}
   Disable_Serial_Devices;

   WriteString('  Serial Protocol Analyzer Hardware
                Verification Check  ' ,Rev);
   WriteStringAt('Checking COM1',HighBlink,2,3);
   port[COM1+LineControl] := TestByte;
   delay(700);
   IF port[COM1+LineControl] <> TestByte THEN
   BEGIN
      WriteStringAt('COM1 Hardware Bad or Missing',
                    Rev,2,3);
      Halt;
   END;
   port[COM1+LineControl] := SafeByte;
   WriteStringAt('COM1 Hardware Verified',
                 Low,2,3);

   delay(700);
   WriteStringAt('Checking COM2',HighBlink,
                 2,4);
   port[COM2+LineControl] := TestByte;
   delay(500);
   IF port[COM2+LineControl] <> TestByte THEN
   BEGIN
      WriteStringAt('COM2 Hardware Bad or Missing',
                    Rev,2,4);
      Halt;
   END;
   port[COM2+LineControl] := SafeByte;
   WriteStringAt('COM2 Hardware Verified',Low,2,4);

   delay(2000);
   ClrScr;

END;


{****** Menu Command Processing Procedure *******}

{Declared forward previously}

PROCEDURE ProcessCmd;
```

Listing One *(Listing continued, text begins on page 30.)*

```
VAR

 HexStr:    Str3;
 HexChar,
 ErrCode:   Integer;
 Ch:        Char;

BEGIN

   IF cmd_code <> 0 THEN
   BEGIN
      CASE cmd_code OF
         1: ExitMenu := True;
         2: BEGIN   {300 baud}
               COM_Rate := 6;
               SetNewCOMParameter;
            END;

         3: BEGIN   {600 baud}
               COM_Rate := 7;
               SetNewCOMParameter;
            END;

         4: BEGIN   {1200 baud}
               COM_Rate := 8;
               SetNewCOMParameter;
            END;

         5: BEGIN   {2400 baud}
               COM_Rate := 11;
               SetNewCOMParameter;
            END;

         6: BEGIN   {4800 baud}
               COM_Rate := 13;
               SetNewCOMParameter;
            END;

         7: BEGIN   {9600 baud}
               COM_Rate := 15;
               SetNewCOMParameter;
            END;

         8: BEGIN   {1 stop bit}
               COM_StopBits := 1;
               SetNewCOMParameter;
            END;

         9: BEGIN   {2 stop bit}
               COM_StopBits := 2;
               SetNewCOMParameter;
            END;

        10: BEGIN   {5 bit word}
               COM_DataBits := 5;
               SetNewCOMParameter;
            END;

        11: BEGIN   {6 bit word}
               COM_DataBits := 6;
               SetNewCOMParameter;
            END;

        12: BEGIN   {7 bit word}
               COM_DataBits := 7;
               SetNewCOMParameter;
            END;

        13: BEGIN   {8 bit word}
               COM_DataBits := 8;

               SetNewCOMParameter;
            END;

        14: BEGIN   {Odd Parity}
               COM_Parity := Odd;
               SetNewCOMParameter;
            END;

        15: BEGIN   {Even Parity}
               COM_Parity := Even;
               SetNewCOMParameter;
            END;

        16: BEGIN   {No Parity}
               COM_Parity := None;
               SetNewCOMParameter;
            END;

        17: BEGIN   {Display COM1 only}
               COM1_Display_Data := True;
               COM2_Display_Data := False;
            END;

        18: BEGIN   {Display COM2 only}
               COM1_Display_Data := False;
               COM2_Display_Data := True;
            END;

        19: BEGIN   {Display both COM1 and COM2}
               COM1_Display_Data := True;
               COM2_Display_Data := True;
            END;

        20: BEGIN   {COM1 is triggered}
               GoToXY(1,25);
               ClrEol;
               COM1_Is_Triggered := True;
               COM2_Is_Triggered := False;
               WriteStringAt('COM1 awaiting trigger',
                             HighBlink,3,25);
            END;

        21: BEGIN   {COM2 is triggered}
               GoToXY(1,25);
               ClrEol;
               COM1_Is_Triggered := False;
               COM2_Is_Triggered := True;
               WriteStringAt('COM2 awaiting trigger',
                             HighBlink,3,25);
            END;

        22: BEGIN   {Input trigger pattern}
               GoToXY(1,MenuLine4);  {position cursor}
               ClrEol;               {clear line}
               WriteString(
                'Input trigger pattern as two hex
                  digits: ',High);

               {initialize str with hex prefix}
               HexStr := '$';
               FOR HexChar := 1 TO 2 DO
               BEGIN
                  REPEAT
                     {get a char}
                     Ch := upcase(Char(GetKey));
                     {loop until valid char is input}
                  UNTIL pos(Ch,HexDigits) <> 0;
                  {display to user}
                  write(Ch);
                  {add to hex string}
                  HexStr := HexStr + Ch;
               END;
               {convert hex to int}
               Val(HexStr,TriggerPattern,ErrCode);
               GoToXY(28,25);
               Write('Trigger Pattern: ',
                     FormatCharHex(TriggerPattern));
            END;

        23: BEGIN   {Display BEFORE Trigger mode}
               TriggerMode := Before;
               WriteStringAt('Mode: Display Before Trigger'
                             ,Low,51,25);
               IF COM1_Is_Triggered THEN
                  {start with displaying data}
                  COM1_Display_Data := True
               ELSE
               IF COM2_Is_Triggered THEN
                  COM2_Display_Data := True
               ELSE
                  WriteStringAt('Mode Error --
                                        Select channel'
                                ,HighBlink,51,25);
            END;

        24: BEGIN   {Display AFTER Trigger mode}
               TriggerMode := After;
               WriteStringAt('Mode: Display After Trigger'
                             ,Low,51,25);
               IF COM1_Is_Triggered THEN
                  {start without displaying data}
                  COM1_Display_Data := False
               ELSE
               IF COM2_Is_Triggered THEN
                  COM2_Display_Data := False
               ELSE
               WriteStringAt('Mode Error -- Select channel'
                             ,HighBlink,51,25);
            END;
```

(continued on page 59)

Listing One *(Listing continued, text begins on page 30.)*

```
      25: TriggerEnabled := True; {enable trigger}

      26: BEGIN   {Stop Triggering}
             TriggerEnabled := False;
             {stop triggering for both channels}
             COM1_Is_Triggered := False;
             COM2_Is_Triggered := False;

             {start data display for both channels}
             COM1_Display_Data := True;
             COM2_Display_Data := True;
             GoToXY(3,25);
             ClrEol;
             WriteString('Triggering Disabled',Low);
          END;

      27: BEGIN   {Ascii Normal Data Display}
             AsciiDisplay := True;
             HandShakeDisplay := False;
          END;

      28: BEGIN   {Ascii with handshake Data Display}
             AsciiDisplay := True;
             HandShakeDisplay := True;
          END;

      29: BEGIN   {Hex Normal Data Display}
             AsciiDisplay := False;
             HandShakeDisplay := False;
          END;

      30: BEGIN   {Hex with handshake Data Display}
             AsciiDisplay := False;
             HandShakeDisplay := True;
          END;

      31: BEGIN   {Toggle adding spaces to display}
             AddSpace := NOT AddSpace;
          END;

      32: BEGIN   {Start data acquire}
             COM1_Data_Acquire := True;
             COM2_Data_Acquire := True;
          END;

      33: BEGIN   {Stop data acquire}
             COM1_Data_Acquire := False;
             COM2_Data_Acquire := False;
          END;

      34: BEGIN   {Clear the display}
             {claim the screen}
             Alloc(ScreenAccess);
             ClrScr;
             DrawMenuFrame;
             {set cursor postion to home}
             OldXPos := 1;
             OldYPos := 1;
             Dealloc(ScreenAccess);
          END;

      35: Init_Program; {reset the analyzer}

      36: BEGIN   {End the analyzer program}
             ExitMenu := True;
             ExitProgram := True;
          END;


    END;
  END;
END;
{************ Begin Task Procedures *************}

PROCEDURE ProcessKeysTask;

VAR

 Ch:   Char;

 Done,
 OldCOM1Flag,
 OldCOM2Flag:  Boolean;

BEGIN

   Done := False;

   REPEAT

      {read key if available else yield}

      Ch := Char(GetKey);

      CASE Ch OF

      Esc: BEGIN  {if key is Esc}
             {display and process the menu}
             Done := DoMenu;
             {rebuild display when finished}
             BuildDisplay;
           END;

      Sp: BEGIN  {if space bar}
             {save old state}
             OldCOM1Flag := COM1_Display_Data;
             OldCOM2Flag := COM2_Display_Data;

             {stop filling of the display fifo so}
             {it does not overflow durin pause}
             {turn off display data both channels}
             COM1_Display_Data := False;
             COM2_Display_Data := False;
             GoToXY(1,24);
             ClrEol;
             GoToXY(23,24);
             WriteString(
               'Press <Enter> to restart the display'
               ,HighBlink);
             REPEAT
                {loop until Cr starts display again}
                Ch := Char(GetKey);
             UNTIL Ch = Cr;

            {restore previous display status}
            COM1_Display_Data := OldCOM1Flag;
            COM2_Display_Data := OldCOM2Flag;
            BuildDisplay; {rebuild display}
          END;
      END;

   UNTIL Done;

END;


PROCEDURE MoveCOM1Data;

{Move data from the COM1 input buffer to the}
{COM2 output buffer and to the display buffer}
{if enabled}

VAR

 Sd:   DataRec;
 Dd:   DisplayRec;

BEGIN

   REPEAT
      {yield if no data to move}
      WHILE COM1_Input_Fifo.Ovd.Count = 0 DO
         yield;
      {when data is available move it}

      {turn ints off, read data, turn ints on}
      INLINE($FA);
      GetSerialData(COM1_Input_Fifo,Sd);
      INLINE($FB);

      {store in COM2 output fifo}
      PutSerialData(Sd,COM2_Output_Fifo);
      {if a trigger is enabled}
      IF COM1_Is_Triggered AND
         TriggerEnabled    THEN
      BEGIN
         {and a match is found}
         IF Sd.Data = TriggerPattern THEN
         {indicate match & disable further matches}
         {trigger is single shot event}
         BEGIN
            TriggerEnabled := False;

            WriteStringAt('   COM1 Triggered
                          ,Low,3,25);
           {If displaying before trigger}
           {then stop data display}
           IF TriggerMode = Before THEN
               COM1_Display_Data := False
           ELSE
               {start the data display}
               COM1_Display_Data := True;

         END;
      END;
```

(continued on next page)

Listing One *(Listing continued, text begins on page 30.)*

```
      {if we are displaying data}
      IF COM1_Display_Data THEN
      {move data to the display fifo also}
      BEGIN
         {tagged from this COM device}
         Dd.DR := Sd;
         Dd.Tag := FromCOM1;
         PutDisplayData(Dd,DisplayFifo);
      END;

   UNTIL False;

END;


PROCEDURE MoveCOM2Data;

{Move data from the COM2 input buffer to the}
{COM1 output buffer and to the display buffer}
{if enabled}

VAR

 Sd:   DataRec;
 Dd:   DisplayRec;

BEGIN

   REPEAT
      {yield if no data to move}
      WHILE COM2_Input_Fifo.Ovd.Count = 0 DO
          yield;

      {when data is available move it}

      {ints off, read data, turn ints on}
      INLINE($FA);
      GetSerialData(COM2_Input_Fifo,Sd);
      INLINE($FB);

      {store data in COM1 output fifo}
      PutSerialData(Sd,COM1_Output_Fifo);

      {if a trigger is enabled}
      IF COM2_Is_Triggered AND
         TriggerEnabled    THEN
      BEGIN
         {indicate match & disable further matches}
         {trigger is single shot event}
         IF Sd.Data = TriggerPattern THEN
         BEGIN
            TriggerEnabled := False;
            WriteStringAt('   COM2 Triggered
                          Low,3,25);
            {If displaying before trigger}
            {stop data display}
            IF TriggerMode = Before THEN
               COM2_Display_Data := False
            ELSE
               {If displaying after trigger}
               {start data display}
               COM2_Display_Data := True;

         END;
      END;
      {if we are displaying data}
      IF COM2_Display_Data THEN
      BEGIN
         {move data to the display fifo also}
         Dd.DR := Sd;

         {tagged from this COM device}
         Dd.Tag := FromCOM2;
         PutDisplayData(Dd,DisplayFifo);
      END;

   UNTIL False;

END;


PROCEDURE OutputCOM1Data;

{Move serial data from the output fifo}
{to the COM port}
```

Listing One *(Listing continued, text begins on page 30.)*

```
CONST

 {max of 80 chars to serial port}
 {between yields to other tasks}

 OutputsPerYield = 80;


VAR

 Outs:   Integer;
 Sd:     DataRec;
 Urgent: Boolean;

BEGIN

   REPEAT

      {do until data is available in fifo}
      WHILE (COM1_Output_Fifo.Ovd.Count = 0) DO
      BEGIN
         {always make the handshake lines}
         {between COM1 and COM2 agree then yield}
         MakeHandshake(COM1,COM2_Status);
         yield;
      END;

      {we have data to output}
      {Its urgent if fifo is over half full}

      Urgent := (COM1_Output_Fifo.Ovd.Count >=
                      (SerialDataFifoSize/2));

      {Initialize output counter to max value}
      Outs := OutputsPerYield;

      {While there is data to output}

      WHILE ((COM1_Output_Fifo.Ovd.Count <> 0) AND
             (Outs <> 0)) DO
      BEGIN

         {test if UART is ready to transmit}

         WHILE (Get_Serial_Status(COM1) AND
                TxRdyBit) = 0 DO
         BEGIN
            {if we have time yield until ready}
            IF NOT Urgent THEN
               yield;
         END;

         {get data record to output}
         {set the handshake lines and output}
         {the data}

         GetSerialData(COM1_Output_Fifo,Sd);
         MakeHandshake(COM1,Sd.Status);
         port[COM1] := Sd.Data;

         {one less to output}
         Outs := Outs - 1;

      END;
      yield;

   UNTIL False;

END;
PROCEDURE OutputCOM2Data;

{Move serial data from the output fifo}
{to the COM port}

CONST

 {max of 80 chars to serial port}
 {between yields to other tasks}

 OutputsPerYield = 80;


VAR

 Outs:   Integer;
 Sd:     DataRec;
 Urgent: Boolean;

BEGIN

   REPEAT

      {do until data is available in fifo}
```

```
      WHILE (COM2_Output_Fifo.Ovd.Count = 0) DO
      BEGIN
         {always make the handshake line}
         {between COM1 and COM2 agree then yield}
         MakeHandshake(COM2,COM1_Status);
         yield;
      END;

      {we have data to output}
      {Its urgent if fifo is over half full}

      Urgent := (COM2_Output_Fifo.Ovd.Count >=
                 (SerialDataFifoSize/2));

      {Initialize output counter to max value}
      Outs := OutputsPerYield;

      {While there is data to output}

      WHILE ((COM2_Output_Fifo.Ovd.Count <> 0) AND
            (Outs <> 0)) DO
      BEGIN
         {test if UART is ready to transmit}

         WHILE (Get_Serial_Status(COM2) AND
                TxRdyBit) = 0 DO
         BEGIN
            {if we have time yield until ready}
            IF NOT Urgent THEN
               yield;
         END;

         {get data record to output}
         {set the handshake lines and output}
         {the data}

         GetSerialData(COM2_Output_Fifo,Sd);
         MakeHandshake(COM2,Sd.Status);
         port[COM2] := Sd.Data;

         {one less to output}
         Outs := Outs - 1;

      END;
      yield;

   UNTIL False;

END;


PROCEDURE DisplayCOMData;

CONST

 {max # of items displayed before yielding}

 MaxDisplayItems = 20;

VAR

 DisplayItems,
 DataLen:      Integer;
 D:            DisplayRec;
 BEGIN

    {initial cursor is homed}
    {in data display window}
    OldXPos := 1;
    OldYPos := 1;

    REPEAT
       {if data in display fifo}
       IF DisplayFifo.Ovd.Count <> 0 THEN
       BEGIN
          {claim the screen, define the window}
          {position the cursor and initialize}
          {items counter}
          Alloc(ScreenAccess);
          Window(1,7,80,23);
          GoToXY(OldXPos,OldYPos);
          DisplayItems := MaxDisplayItems;

          WHILE ((DisplayFifo.Ovd.Count <> 0 ) AND
                 (DisplayItems <> 0)) DO
          {while there is data to display}
          BEGIN
             {get the data, display it, dec}
             {item counter. Save len of displayed}
             {item.}

             GetDisplayData(DisplayFifo,D);
             DataLen := DisplayData(D);
             DisplayItems := DisplayItems - 1;

             {if not room to display for another}
             {item of same length and we're on the}
             {last display line in window, then}
             {home cursor and start overwriting}
             {screen so as to avoid the scroll of}
             {the display. If not on last line of}
             {window, advance to next line. Either}
             {way, the next line must be cleared.}

             IF WhereX + DataLen >= 80 THEN
                IF WhereY = 17 THEN
                   GoToXY(1,1)
                ELSE
                   WriteLn;

             ClrEol;
          END;

          {save new cursor position}
          OldXPos := WhereX;
          OldYPos := WhereY;

          {back to full screen}
          {and release screen lock and yield}
          Window(1,1,80,25);
          Dealloc(ScreenAccess);
       END
       ELSE
          {if no data to display just yield}
          yield;

    UNTIL False;

 END;


 PROCEDURE Timer;

 VAR

  Toggle:  Boolean;

 BEGIN

 Toggle := True;

 REPEAT

    {every 1/2 second do the following}
    pause(4);

    {if ok to update the screen}
    IF UpdateScreenStatus THEN
    BEGIN
       {lock the screen}
       Alloc(ScreenAccess);

       {if status of either port has changed}
       IF ((COM1_Status <> OldCOM1_Status)  OR
           (COM2_Status <> OldCOM2_Status)) THEN

       {then update the screen}
       BEGIN
          {display new COM1 status}
          DisplayHandShakeStatus(COM1,COM1_Status
                                 ,COM2_Status);
          {save new status}
          OldCOM1_Status := COM1_Status;

          {display new COM2 status}
          DisplayHandShakeStatus(COM2,COM2_Status
                                 ,COM1_Status);
          OldCOM2_Status := COM2_Status;

       END;

       {every full second update}
       {buffer status on screen}
       IF Toggle THEN
          DisplayBufferStatus;

       {unlock the screen}
       Dealloc(ScreenAccess);
       Toggle := NOT Toggle;
    END;

 UNTIL False;

END;
```

Listing One *(Listing continued, text begins on page 30.)*

```
BEGIN {main}

   {verify presence of COM1 and COM2}
   VerifyHardware;
   {initialize the multitasker}
   Init_Kernel;
   {initialize the menu structure}
   Init_Menu;
   {install the serial ISRs}
   Install_Serial_Handlers;

   {Initialize strings used to format the display}
   Line1 := Char(201)+' COM1 '+repl(16,Char(205))+
          ' Serial Protocol Analyzer Ver: 1.0 '+
          repl(15,Char(205))+ ' COM2 '+Char(187);

   Line2 := Char(186)+' In Fifo:'+repl(5,' ')+
            'Out Fifo:    '+ Char(186)+
            repl(10,' ')+'By'+repl(10,' ')+
            Char(186)+' In Fifo:'+repl(5,' ')+
            'Out Fifo:    '+Char(186);

   Line3 := Char(186)+' Stat- BRK:  FE:  PE:  OR: '+
            Char(186)+ '   Craig A. Lindley   '+
            Char(186)+ ' Stat- BRK:  FE:  PE:  OR: '+
            Char(186);

   Line4 := Char(186)+' In - CD:  RI:  DSR:  CTS: '+
            Char(186)+repl(22,' ')+Char(186)+
            ' In - CD:  RI:  DSR:  CTS: '+Char(186);

   Line5 := Char(186)+ ' Out- DTR:      RTS:'+
            repl(7,' ')+Char(186)+
            '   Display Fifo:      '+Char(186)+
            ' Out- DTR:      RTS:'+repl(7,' ')+
            Char(186);

   Line6 := Char(200)+repl(27,Char(205))+
            Char(202)+repl(22,Char(205))+
            Char(202)+repl(27,Char(205))+
            Char(188);

   Line24 := '  <Esc> for Menu  <Space Bar> to Pause'+
           '    COM1 - Normal : COM2 - Reverse Video';

   BuildDisplay;   {show main display}
   Init_Program;   {initialize all vars}

   {initialize screen lock}
   Initialize_Semaphore(ScreenAccess);

   {enable status update}
   UpdateScreenStatus:= True;

   {Fork off all tasks}
    Fork;

    IF child_process THEN
       MoveCOM1Data;

    Fork;

    IF child_process THEN
       MoveCOM2Data;

    Fork;

    IF child_process THEN
       OutputCOM1Data;

    Fork;

    IF child_process THEN
       OutputCOM2Data;

    Fork;

    IF child_process THEN
       DisplayCOMData;

    Fork;

    IF child_process THEN
       Timer;


    ProcessKeysTask;

    ClrScr;
    Disable_Serial_Devices;
    Remove_Serial_Handlers;

END
```

End Listing One

Listing Two

```
{*********************************************}
{***                                       ***}
{***       Menu System support procedures  ***}
{***      for the serial protocol analyzer ***}
{***                 written by            ***}
{***              Craig A. Lindley         ***}
{***                                       ***}
{***    Ver: 2.0     Last update: 08/15/87  ***}
{***                                       ***}
{*********************************************}

CONST

 {max # of video attributes - 1}
 AttribMax = 5;

 Attributes: ARRAY[0..AttribMax] OF
 RECORD
    f,              {foreground color}
    b: Integer;     {background color}
 END

 {Attributes are:}
 {Low, High, Rev, LowBlink, HighBlink, RevBlink}

  = ((f:7; b:0) , (f:15;b:0) , (f:0; b:7),
     (f:23;b:0) , (f:31;b:0) , (f:16;b:7));


 {menu display lines relative to screen line 1}
 MenuLine1 = 2;
 MenuLine2 = 3;
 MenuLine3 = 4;
 MenuLine4 = 5;

 {max A tree dimensions}
 level2width = 7;
 level3width = 5;
 level4width = 7;

 home=    #199;    {home key code}
 larrow=  #203;    {left cursor arrow code}
 rarrow=  #205;    {right cursor arrow code}
 endkey=  #207;    {end key code}
 bs=      #08;     {backspace key}
 lf=      #10;     {line feed code}
 cr=      #13;     {carrage return code}
 Esc=     #27;     {escape key code}
 sp=      #32;     {ascii space code}

TYPE

 AttribType  = (Low ,High, Rev, LowBlink,
                HighBlink, RevBlink);

 {data structure for atree menu entry}
 {see text for details}

 menu_entry=
 RECORD
    title:  STRING[10];
     desc:  STRING[40];
    chars:  STRING[8];
    index:  Byte;
    ccode:  Byte;
 END;

tree = ARRAY[0..level2width,
             0..level3width,
             0..level4width]
       OF menu_entry;

VAR

 ExitMenu,
 ExitProgram:      Boolean;

 ind1,ind2,
 ind3,
 selector,
 cmd_code,
 level:            Byte;

 {atree data structure used for nested menus}
 atree:            tree;


{******* Keyboard and Display Procedures ********}

PROCEDURE Beep;
```

(continued on page 68)

Listing Two *(Listing continued text begins on page 30.)*

```
BEGIN

   Sound(1000);
   Pause(1);
   NoSound;

END;


FUNCTION GetKey : Byte;

VAR

 Ch: Char;
 B:  Byte;

 CurrentX,
 CurrentY:  Integer;

BEGIN

  {save cursor position}
  {because we are going to yield}
  {and loose control of display}

  CurrentX := WhereX;
  CurrentY := WhereY;

  {yield until a key is pressed}

  WHILE NOT keypressed DO
      yield ;

  {put the cursor back}
  GoToXY(CurrentX,CurrentY);

  {read the key}
  read(kbd,Ch);

  {see if its an extended key}
  IF (Ch = Esc) AND keypressed THEN
  BEGIN
    {if so read again and mark}
    {as extended by setting MSB}
    read(kbd,Ch);
    B := ord(Ch)+128;
  END
  ELSE
    {in either case B has key code}
    B := ord(Ch);

  GetKey := B;

END;


FUNCTION repl (Count:Integer; ch:Char):FullString;

{replicate a char into a string of}
{specified length}

VAR

 i:               Integer;

BEGIN

   FOR i:=1 TO Count DO repl[i]:=ch;
   repl[0]:=Char(Count);

END;


PROCEDURE WriteString (Stng:FullString;
                       Attrib:AttribType);

BEGIN

   {set the foreground and background}
   {colors and write the string}

   WITH Attributes[ORD(Attrib)] DO
   BEGIN
      TextColor(f);
      TextBackGround(b);
   END;
   write(Stng);

   {go back to low video mode}
   WITH Attributes[ORD(Low)] DO
   BEGIN
      TextColor(f);
      TextBackGround(b);
   END;

END;


PROCEDURE WriteStringAt (Stng:FullString;
                         Attrib:AttribType;
                         X,Y:Integer);

BEGIN

 GoToXY(X,Y);
 WriteString(Stng,Attrib);

END;


{*********** Start of Menu Procedures ***********}

PROCEDURE center (Line,Width:Integer;
                  Outstr:FullString;
                  Attrib:AttribType);

{Center and write string with attribute on a}
{specified line in a given width field}

BEGIN

   GoToXY(1,Line);
   Clreol;
   GoToXY((Width DIV 2)-(length(Outstr) DIV 2),
          Line);
   WriteString(Outstr,Attrib);

END;


PROCEDURE DrawFrame (x,y,width,height:Integer);

VAR

 top,bottom:     Str80;

BEGIN

   top:=   repl(width,Char(205));
   bottom:=repl(width,Char(205));
   WriteStringAt(top,High,x,y);
   WriteStringAt(bottom,High,x,y+height-1);

END;


PROCEDURE DrawMenuFrame;

BEGIN

   DrawFrame(1,1,80,6);
   center(MenuLine1,78,
          ' Serial Protocol Analyzer Menu ',Rev);

END;


PROCEDURE DisplayMenu (ind1,ind2,ind3,level,
                       selector:Byte);

VAR

 tind1,tind2,
 tind3,i:     Integer;
 attrib:      AttribType;
 titlestr:    FullString;

BEGIN
   titlestr:='-- '+
     atree[ind1,ind2,ind3].title+' Selection'+' --';
   center(MenuLine2,78,titlestr,Low);
   GoToXY(1,MenuLine4);
   clreol;
   GoToXY(9,MenuLine3);
   clreol;

   FOR i:= 1 TO atree[ind1,ind2,ind3].index DO
   BEGIN
      CASE level OF
         1: ind1:=i;
         2: ind2:=i;
```

(continued on page 72)

Listing Two *(Listing continued, text begins on page 30.)*

```
            3: ind3:=i;
         END;
         IF selector <> i THEN  {item not selected}
             attrib:=Low
         ELSE
         BEGIN
            attrib:=Rev;
            tind1:=ind1;
            tind2:=ind2;
            tind3:=ind3;
         END;
         WriteString(atree[ind1,ind2,ind3].title,
                     attrib);
         write('   '); {spaces to separate items}
      END;
      GoToXY(9,MenuLine4);
      clreol;
      write(atree[tind1,tind2,tind3].desc);

END;


PROCEDURE ProcessCr (VAR ind1,ind2,ind3,level,
                     selector,cmd_code:Byte);

BEGIN
 {assign selector to index as it was picked}
 CASE level OF
    1: ind1:=selector;
    2: ind2:=selector;
    3: ind3:=selector;
 END;
 {if entry is terminal entry}
 IF atree[ind1,ind2,ind3].index = 0 THEN
 BEGIN
    {get associated cmd code}
    cmd_code:=atree[ind1,ind2,ind3].ccode;

    {process depending upon level}
    CASE level OF
       1: ind1:=0;
       2: BEGIN
             ind1:=0;
             ind2:=0;
             ind3:=0;
             level:=1;
          END;
       3: BEGIN
             ind2:=0;
             ind3:=0;
             level:=2;
          END;
    END;
 END
 ELSE
 BEGIN
    {down to next menu level}
    level:=level+1;
 END;
 {select set to 1st item}
 selector:=1;

END;


PROCEDURE ProcessMenu (VAR ind1,ind2,ind3,level,
                       selector,cmd_code:Byte);

VAR

 key:          Char;
 position:     Integer;

BEGIN

   key:=upcase(GetKey);
   CASE key OF
    rarrow: BEGIN
              selector:=selector+1;
              IF selector >
                 length(atree[ind1,ind2,ind3].chars) THEN
                  selector:=1;
            END;

    larrow: BEGIN
              selector:=selector-1;
              IF selector < 1 THEN
                 selector :=
                length(atree[ind1,ind2,ind3].chars);
            END;

    endkey: selector :=
              length(atree[ind1,ind2,ind3].chars);

      home: selector:=1;

        cr: ProcessCr(ind1,ind2,ind3,level,
                      selector,cmd_code);

       esc: cmd_code := 1;     {force exit}

'A'..'Z','0'..'9'  {1st letter of menu item ?}
         : BEGIN
             position :=
               pos(key,atree[ind1,ind2,ind3].chars);
             IF position <> 0 THEN
             BEGIN             {is 1st letter}
                selector:=position;
                ProcessCr(ind1,ind2,ind3,level,
                          selector,cmd_code);
             END
             ELSE     {is not so beep}
                Beep;
           END;
      ELSE                  {invalid so beep}
          beep;
   END;

END;


{Procedure used to process the operator}
{selected commands}
{Defined in the main program's code}

PROCEDURE ProcessCmd (cmd_code: Byte); FORWARD;


FUNCTION DoMenu : Boolean;

VAR

 Line: Integer;

BEGIN

   {stop update of screen info}
   UpdateScreenStatus := False;

   {clear top 6 display lines}
   FOR Line := 1 TO 6 DO
   BEGIN
      GoToXY(1,Line);
      Clreol;
   END;

   {and 2nd to the last one}
   GoToXY(1,24);
   Clreol;

   DrawMenuFrame;
   {clear the menu exit flag}
   ExitMenu := False;

   {clear the program exit flag}
   ExitProgram := False;

   {selector to 1st item in menu}
   selector:=1;

   {command code initialized to 0 }
   cmd_code:=0;

   {1st tier of heirachy}
   level:=1;

   {root menu array location}
   ind1:=0;
   ind2:=0;
   ind3:=0;

   REPEAT

      DisplayMenu (ind1,ind2,ind3,level,selector);
      ProcessMenu (ind1,ind2,ind3,level,selector,
                   cmd_code);
      ProcessCmd  (cmd_code);
      cmd_code:=0;  {reset code selected last}

   UNTIL ExitMenu;

   {start update of screen info}
```

```
   UpdateScreenStatus := True;
   {return flag}
   DoMenu := ExitProgram;

END;


PROCEDURE Init_Menu;

{initialize the menu tree}

BEGIN

   fillchar(atree,sizeof(atree),0);

   WITH atree[0,0,0] DO
   BEGIN
      title:='Main Menu';
      chars:='QPDTFCE';
      index:=7;
   END;

   WITH atree[1,0,0] DO
   BEGIN
      title:='Quit';
      desc:='Exit Analyzer Menus';
      ccode:=1;
   END;

   WITH atree[2,0,0] DO
   BEGIN
      title:='Parameters';
      desc:='Set Serial Parameters';
      chars:='QBSWP';
      index:=5;
   END;

   WITH atree[2,1,0] DO
   BEGIN
      title:='Quit';
      desc:='Exit this submenu';
   END;

   WITH atree[2,2,0] DO
   BEGIN
      title:='Baud Rate';
      desc:='Change Serial
            Baud Rate';
      chars:='Q361249';
      index:=7;
   END;

   WITH atree[2,2,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this submenu';
   END;

   WITH atree[2,2,2] DO
   BEGIN
      title:='300';
      ccode:=2;
   END;

   WITH atree[2,2,3] DO
   BEGIN
      title:='600';
      ccode:=3;
   END;

   WITH atree[2,2,4] DO
   BEGIN
      title:='1200';
      ccode:=4;
   END;

   WITH atree[2,2,5] DO
   BEGIN
      title:='2400';
      ccode:=5;
   END;

   WITH atree[2,2,6] DO
   BEGIN
      title:='4800';
      ccode:=6;
   END;

   WITH atree[2,2,7] DO
   BEGIN
      title:='9600';
      ccode:=7;
   END;

   WITH atree[2,3,0] DO
   BEGIN
      title:='Stop Bits';
      desc:='Set Number of
            Stop Bits';
      chars:='Q12';
      index:=3;
   END;

   WITH atree[2,3,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this submenu';
   END;

   WITH atree[2,3,2] DO
   BEGIN
      title:='1';
      ccode:=8;
   END;

   WITH atree[2,3,3] DO
   BEGIN
      title:='2';
      ccode:=9;
   END;

   WITH atree[2,4,0] DO
   BEGIN
      title:='Word Len.';
      desc:='Set Bits / Word';
      chars:='Q5678';
      index:=5;
   END;

   WITH atree[2,4,1] DO
   BEGIN
     title:='Quit';
     desc:='Exit this SubMenu';
   END;

   WITH atree[2,4,2] DO
   BEGIN
      title:='5';
      ccode:=10;
   END;

   WITH atree[2,4,3] DO
   BEGIN
      title:='6';
      ccode:=11;
   END;

   WITH atree[2,4,4] DO
   BEGIN
      title:='7';
      ccode:=12;
   END;

   WITH atree[2,4,5] DO
   BEGIN
      title:='8';
      ccode:=13;
   END;

   WITH atree[2,5,0] DO
   BEGIN
      title:='Parity';
      desc:='Set Serial Parity';
      chars:='QOEN';
      index:=4;
   END;

   WITH atree[2,5,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[2,5,2] DO
   BEGIN
      title:='Odd';
      ccode:=14;
   END;

   WITH atree[2,5,3] DO
   BEGIN
      title:='Even';
      ccode:=15;
   END;

   WITH atree[2,5,4] DO
   BEGIN
      title:='None';
      ccode:=16;
   END;

   WITH atree[3,0,0] DO
   BEGIN
     title:='Display';
     desc:='Select Channels
           for Display';
     chars:='Q12B';
     index:=4;
   END;

   WITH atree[3,1,0] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[3,2,0] DO
   BEGIN
      title:='COM1';
      ccode:=17;
   END;

   WITH atree[3,3,0] DO
   BEGIN
      title:='COM2';
      ccode:=18;
   END;

   WITH atree[3,4,0] DO
   BEGIN
      title:='Both 1 & 2';
      ccode:=19;
   END;

   WITH atree[4,0,0] DO
   BEGIN
      title:='Trigger';
      desc:='Controls
            Trigger Modes';
      chars:='QCPMS';
      index:=5;
   END;

   WITH atree[4,1,0] DO
   BEGIN
     title:='Quit';
     desc:='Exit this SubMenu';
   END;

   WITH atree[4,2,0] DO
   BEGIN
      title:='Channel';
      desc:='Set Trigger Channel';
      chars:='Q12',
      index:=3;
   END;

   WITH atree[4,2,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[4,2,2] DO
   BEGIN
      title:='COM1';
      ccode:=20;
   END;

   WITH atree[4,2,3] DO
   BEGIN
      title:='COM2';
      ccode:=21;
   END;

   WITH atree[4,3,0] DO
   BEGIN
      title:='Pattern';
      desc:='Input Pattern

      ccode:=22;
   END;

   WITH atree[4,4,0] DO
   BEGIN
      title:='Mode';
      desc:='Select Trigger Mode';
      chars:='QBAE';
      index:=4;
   END;

   WITH atree[4,4,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[4,4,2] DO
   BEGIN
      title:='Before';
      desc:='Display Data Before Trigger';
      ccode:=23;
   END;

   WITH atree[4,4,3] DO
   BEGIN
      title:='After';
      desc:='Display Data After Trigger';
      ccode:=24;
   END;

   WITH atree[4,4,4] DO
   BEGIN
      title:='Enable';
      desc:='Enable the Trigger';
      ccode:=25;
   END;

   WITH atree[4,5,0] DO
   BEGIN
      title:='Stop';
      desc:='Stop waiting for Trigger Event';
      ccode:=26;
   END;

   WITH atree[5,0,0] DO
   BEGIN
      title:='Format';
      desc:='Set Display Format';
      chars:='QAHS';
      index:=4;
   END;

   WITH atree[5,1,0] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[5,2,0] DO
   BEGIN
      title:='Ascii';
      desc:='Ascii Display Format';
      chars:='QNH';
      index:=3;
   END;

   WITH atree[5,2,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
   END;

   WITH atree[5,2,2] DO
   BEGIN
      title:='Normal';
      desc:='Data Only';
      ccode:=27;
   END;

   WITH atree[5,2,3] DO
   BEGIN
      title:='HandShake';
      desc:='Data and HandShake';
      ccode:=28;
   END;

   WITH atree[5,3,0] DO
   BEGIN
      title:='Hex';
      desc:='Hex Display Format';
      chars:='QNH';
      index:=3;
   END;

   WITH atree[5,3,1] DO
   BEGIN
      title:='Quit';
      desc:='Exit this SubMenu';
```

Listing Two *(Listing continued, text begins on page 30.)*

```
  END;

  WITH atree[5,3,2] DO
  BEGIN
     title:='Normal';
     desc:='Data Only';
     ccode:=29;
  END;

  WITH atree[5,3,3] DO
  BEGIN
     title:='HandShake';
     desc:='Data and HandShake';
     ccode:=30;
  END;

  WITH atree[5,4,0] DO
  BEGIN
     title:='Spaces';
     desc:='Insert space in data display';
     ccode:=31;
  END;

  WITH atree[6,0,0] DO
  BEGIN
     title:='Control';
     desc:='Alter Analyzer Operation';
     chars:='QASCR';
     index:=5;
  END;

  WITH atree[6,1,0] DO
  BEGIN
     title:='Quit';
     desc:='Exit this SubMenu';
  END;

  WITH atree[6,2,0] DO
  BEGIN
     title:='Acquire';
     desc:='Acquire Data';
     ccode:=32;
  END;

  WITH atree[6,3,0] DO
  BEGIN
     title:='Stop';
     desc:='Stop Data Acquisition';
     ccode:=33;
  END;

  WITH atree[6,4,0] DO
  BEGIN
     title:='Clear';
     desc:='Clear Screen of Data';
     ccode:=34;
  END;
  WITH atree[6,5,0] DO
  BEGIN
     title:='Reset';
     desc:='Reset Analyzer';
     ccode:=35;
  END;

  WITH atree[7,0,0] DO
  BEGIN
     title:='End';
     desc:='End the Analyzer Session';
     ccode:=36;
  END;
END;
```

End Listing Two

Listing Three

```
{***********************************************}
{***                                         ***}
{***          RS-232 support procedures      ***}
{***      for the serial protocol analyzer   ***}
{***                written by               ***}
{***             Craig A. Lindley            ***}
{***                                         ***}
{***    Ver: 2.0     Last update: 08/15/87   ***}
{***                                         ***}
{***********************************************}

CONST

  COM1 = $3f8;              {com one port addr}
  COM2 = $2f8;              {com two port addr}

  {table of values for the various baud rates}
  {supported by the 8250. Rates from 50..9600}

  BaudRate: ARRAY[1..15] OF Integer =
    ($900,$600,$417,$359,$300,$180,$0C0,$060,
     $040,$03A,$030,$020,$018,$010,$00C);

  {8259 registers}

  Int_Mask_Reg = $21; {interrupt enable register}
  Int_Cmd_Reg  = $20; {command register}
  End_Int_Cmd  = $20; {end of interrupt cmd}

  {offsets from PortAddr for the various}
  {8250 registers}

  DLL             = 0;
  DLM             = 1;
  Int_Enable_Reg  = 1;
  Int_Id_Reg      = 2;
  LineControl     = 3;
  ModemControl    = 4;
  LineStatus      = 5;
  ModemStatus     = 6;

  {Status bit definitions}

  DataRdyBit      = $01;
  DTRBit          = $01;
  Out2Bit         = $08;
  ORBit           = $01;
  RTSBit          = $02;
  PEBit           = $02;
  FEBit           = $04;
  BrkBit          = $08;
  CTSBit          = $10;
  DSRBit          = $20;
  TxRdyBit        = $20;
  RIBit           = $40;
  CDBit           = $80;

TYPE

  ParityType = (odd,even,none);

VAR

  {Global storage of COM parameters}
  {Used by the SetNewCOMParameter}
  {procedure}

  COM_Rate,
  COM_StopBits,
  COM_DataBits:  Integer;
  COM_Parity:    ParityType;


{************** Serial Procedures ***************}

PROCEDURE Enable_Serial_Device (PortAddr:Integer);

VAR

  Temp:   Byte;

BEGIN

   {clear the 8250 serial device of any garbage}
   {by reading data port, int id reg. line}
   {status reg and modem status reg}

    Temp := port[PortAddr];
    Temp := port[PortAddr+Int_Id_Reg];
    Temp := port[PortAddr+LineStatus];
    Temp := port[PortAddr+ModemStatus];

    {read 8259 int mask reg and set IRQ3 or IRQ4}
    {low to enable requested interrupts.}
    {write result back to 8259 when finished}

    Temp := port[Int_Mask_Reg];
    IF PortAddr = COM1 THEN
       Temp := Temp AND $EF
    ELSE
       Temp := Temp AND $F7;
    port[Int_Mask_Reg] := Temp;

    {Out2, DTR and RTS high for 8250}
    port[PortAddr+ModemControl] :=
         Out2Bit + DTRBit + RTSBit;
```

(continued on next page)

Listing Three *(Listing continued, text begin on page 30.)*

```
    {all ints active except Tx}
    port[PortAddr+Int_Enable_Reg] := $0D;
    delay(100);

END;


PROCEDURE Disable_Serial_Devices;

VAR

 Temp:   Byte;

BEGIN

   {Read int mask in 8259, set both IRQ3}
   {and IRQ4 bits high to disable. Set Out2}
   {low for both COM1 and COM2 to prevent ints}
   {from leaving the async card and finally}
   {disable all interrupt sources in the UART}

   Temp := port[Int_Mask_Reg];
   Temp := Temp OR $18;
   port[Int_Mask_Reg] := Temp;
   port[COM1+ModemControl] := 0;
   port[COM2+ModemControl] := 0;
   port[COM1+Int_Enable_Reg] := 0;
   port[COM2+Int_Enable_Reg] := 0;

END;


PROCEDURE COM1_ISR;

{COM1 interrupt service routine}

VAR

 SerialInfo:  DataRec;

 Temp:        Integer;

 LineState,
 ModemState:  Byte;

BEGIN

   {read linestatus and modem status and}
   {combine into COM1_Status. See text for}
   {bit encoding.}

   LineState := port[COM1+LineStatus];
   ModemState:= port[COM1+ModemStatus];
   Temp := (LineState SHR 1) AND $0F;

   {COM1_Status has the UART state in 8 bits}
   {this is comprised of status info only, no data}

   COM1_Status := (ModemState AND $F0) OR lo(Temp);

   {if there is data to receive and we are}
   {acquiring data}

   IF (((LineState AND DataRdyBit) <> 0) AND
        COM1_Data_Acquire) THEN
   BEGIN
      {if there is room in the COM1 input fifo}
      IF COM1_Input_Fifo.Ovd.Count <
                    SerialDataFifoSize THEN
      BEGIN
         {Put received data and status into fifo}
         SerialInfo.Data   := port[COM1];
         SerialInfo.Status := COM1_Status;
         PutSerialData(SerialInfo,COM1_Input_Fifo);
      END
      ELSE
      BEGIN
         writeln('COM1 Input fifo overflowed');
         halt;
      END;
   END;

   {signal end of int to 8259}
   port[Int_Cmd_Reg] := End_Int_Cmd;

END;


PROCEDURE COM2_ISR;

{COM2 interrupt service routine}

VAR

 SerialInfo:  DataRec;

 Temp:        Integer;

 LineState,
 ModemState:  Byte;

BEGIN

   {read linestatus and modem status and}
   {combine into COM2_Status. See text for}
   {bit encoding.}

   LineState := port[COM2+LineStatus];
   ModemState:= port[COM2+ModemStatus];
   Temp := (LineState SHR 1) AND $0F;

   {COM2_Status has the UART state in 8 bits}
   {this is comprised of status info only, no data}

   COM2_Status := (ModemState AND $F0) OR lo(Temp);

   {if there is data to receive and we are}
   {acquiring data}

   IF (((LineState AND DataRdyBit) <> 0) AND
        COM2_Data_Acquire) THEN
   BEGIN
      {if there is room in the COM2 input fifo}
      IF COM2_Input_Fifo.Ovd.Count <
                    SerialDataFifoSize THEN
      BEGIN
        {Put received data and status into fifo}
        SerialInfo.Data   := port[COM2];
        SerialInfo.Status := COM2_Status;
        PutSerialLata(SerialInfo,COM2_Input_Fifo);
      END
      ELSE
      BEGIN
         writeln('COM2 Input fifo overflowed');
         halt;
      END;
   END;

   {signal end of int to 8259}
   port[Int_Cmd_Reg] := End_Int_Cmd;

END;


PROCEDURE COM1_Int_Service_Routine;

BEGIN

 INLINE($50/$53/$51/$52/$57/   {Push ax,bx,cx,dx,}
        $56/$06/$1e/           {di,si,es,ds}
        $2e/$a1/turbodseg/     {mov ax,cs:turbodseg}
        $8e/$d8/               {mov ds,ax}
        $fb);                  {sti}

 COM1_ISR;

 {standard interrupt service routine postamble}

 INLINE($fa/$1f/$07/$5e/$5f/   {interrupts off}
        $5a/$59/$5b/$58/       {Pop ds,es,si,di,}
                               {dx,cx,bx,ax}
        $5d/$5d/$cf);          {trash sp, restore}
                               {Bp and iret}
 END;


 PROCEDURE COM2_Int_Service_Routine;

 BEGIN

  INLINE($50/$53/$51/$52/$57/   {Push ax,bx,cx,dx,}
         $56/$06/$1e/           {di,si,es,ds}
         $2e/$a1/turbodseg/     {mov ax,cs:turbodseg}
         $8e/$d8/               {mov ds,ax}
         $fb);                  {sti}

  COM2_ISR;

  {standard interrupt service routine postamble}
```

(continued on next page)

Listing Three *(Listing continued, text begins on page 30.)*

```
  INLINE($fa/$1f/$07/$5e/$5f/    {interrupts off}
         $5a/$59/$5b/$58/        {Pop ds,es,si,di,}
                                 {dx,cx,bx,ax}
         $5d/$5d/$cf);           {trash sp, restore}
                                 {Bp and iret}

END;


PROCEDURE Install_Serial_Handlers;

BEGIN

 WITH regs DO      {install into IRQ3 & 4}
 BEGIN
    ah := $35;        {get vector func. code}
    al := $0B;        {for IRQ3}
    msdos(regs);      {call dos to get vector}
    OldIRQ3_CS := es;{save code seg}
    OldIRQ3_IP := bx;{and instruction ptr}

    ah := $35;        {get vector func. code}
    al := $0C;        {for IRQ4}
    msdos(regs);      {call dos to get vector}
    OldIRQ4_CS := es;{save code seg}
    OldIRQ4_IP := bx;{and instruction ptr}

    ah := $25;        {set vector func. code}
    al := $0C;        {for IRQ4}
    ds := cseg;       {code in our segment}
                      {at this offset}
    dx := ofs(COM1_Int_Service_Routine);
    msdos(regs);      {call dos to set vector}

    ah := $25;        {set vector func. code}
    al := $0B;        {for IRQ3}
    ds := cseg;       {code in our segment}
                      {at this offset}
    dx := ofs(COM2_Int_Service_Routine);
    msdos(regs);      {call dos to set vector}

 END;
END;


PROCEDURE Remove_Serial_Handlers;

BEGIN
   {Put saved vectors for IRQ3 and IRQ4 back}
   WITH regs DO
   BEGIN
      ah := $25;
      al := $0C;
      ds := OldIRQ4_CS;
      dx := OldIRQ4_IP;
      msdos(regs);

      ah := $25;
      al := $0B;
      ds := OldIRQ3_CS;
      dx := OldIRQ3_IP;
      msdos(regs);
   END;

END;


PROCEDURE Set_Serial_Parameters
     (PortAddr,Baud,StopBits,DataBits:Integer;
      Parity: ParityType);
VAR

 Temp,
 Rate:   Integer;

BEGIN

   {set DLAB high for divisor regs}
   port[PortAddr+LineControl] := $80;

   {look up rate word in table}
   Rate := BaudRate[Baud];

   {MSB into most sign divisor reg}
   port[PortAddr+DLM] := hi(Rate);

   {LSB into less sign divisor reg}
   port[PortAddr+DLL] := lo(Rate);

   {move databits into 2 least sign bits}
   {add in stop bit into bit pos 2}
   {set parity enable and parity even}
   {bits if appropriate}

   Temp := (DataBits - 5) AND $03;
   Temp := Temp OR ((StopBits - 1) SHL 2);
   CASE Parity OF
     odd:  Temp := Temp + $08;
    even: Temp := Temp + $18;
    none: ;
   END;

   {remove DLAB and setup parameters}
   port[PortAddr+LineControl] := lo(Temp);
   delay(100);

END;


FUNCTION Get_Serial_Status (PortAddr:Integer)
             :Integer;

VAR

 Temp:  Integer;

BEGIN

   {Get full 16 bits of COM port status}
   {grouped as ModemStatus:LineStatus}

   Temp := port[PortAddr+ModemStatus];
   Temp := Temp SHL 8;
   Temp := Temp OR port[PortAddr+LineStatus];
   Get_Serial_Status := Temp;

END;


PROCEDURE SetNewCOMParameter;

BEGIN

   {take the COM ports down}
   Disable_Serial_Devices;

   {Set the COM ports to the global parameters}

   Set_Serial_Parameters(COM1,COM_Rate,
         COM_StopBits,COM_DataBits,COM_Parity);
   Set_Serial_Parameters(COM2,COM_Rate,
         COM_StopBits,COM_DataBits,COM_Parity);

   {bring COM ports back up}
   Enable_Serial_Device(COM1);
   Enable_Serial_Device(COM2);

END;
```

End Listing Three

Listing Four

```
{$K-}             {Compiler switch - never change}

{***********************************************}
{***                                         ***}
{***                Turbo Pascal             ***}
{***            Multitasking Kernel          ***}
{***                 written by              ***}
{***              Craig A. Lindley           ***}
{***                                         ***}
{***    Ver: 2.0     Last update: 08/15/87    ***}
{***                                         ***}
{***********************************************}

CONST

   task_stack_size  = 1000;
PROCEDURE Yield;

{This version in assembly language for}
{speed. See version above for comments.}

BEGIN

   IF cp^.link <> cp THEN  {must have more than}
                           {one task forked to be}
                           {able to yield}
```

```
    BEGIN
       INLINE($C6/$06/child_process/$00/
                              {child_process is false}
              $C4/$3E/cp/      {les di,[cp]}
              $89/$E0/         {mov ax,sp}
              $05/$02/$00/     {add ax,2}
              $26/             {es:}
              $89/$45/$04/     {mov [di+4],ax}
              $26/             {es:}
              $C6/$45/$06/$00/ {mov byte ptr [di+6],0}
              $89/$FB/         {L1: mov bx,di}
              $26/             {es:}
              $C4/$1F/         {les bx,[bx]}
              $26/             {es:}
              $80/$7F/$06/$00/ {cmp byte ptr [bx+6],0}
              $74/$04/         {je L2}
              $89/$DF/         {mov di,bx}
              $EB/$F0/         {jmp L1}
              $89/$1E/cp/      {L2: mov [cp],bx}
              $8C/$06/cp+2/    {    mov [cp+2],es}
              $26/             {es:}
              $C6/$47/$06/$02/ {mov byte ptr [bx+6],2}
              $26/             {es:}
              $8B/$6F/$04);    {mov bp,[bx+4]}
    END
    ELSE
    BEGIN
       writeln('Cannot yield only single task running');
       halt;
    END;

END;

PROCEDURE Wait;  {put current task in wait mode}
                 {until a send makes it ready}

BEGIN

   IF cp^.link <> cp THEN  {must have more than}
                           {one task forked to be}
                           {able to wait}
   BEGIN

      waitfor^ := cp;      {waitfor points at the}
                           {current task}
   INLINE($C6/$06/child_process/$00/
                           {child_process is false}
          $C4/$3E/cp/      {les di,[cp]}
          $89/$E0/         {mov ax,sp}
          $05/$02/$00/     {add ax,2}
          $26/             {es:}
          $89/$45/$04/     {mov [di+4],ax}
          $26/             {es:}
          $C6/$45/$06/$01/ {mov byte ptr [di+6],1}
          $89/$FB/         {L1: mov bx,di}
          $26/             {es:}
          $C4/$1F/         {les bx,[bx]}
          $26/             {es:}
          $80/$7F/$06/$00/ {cmp byte ptr [bx+6],0}
          $74/$04/         {je L2}
          $89/$DF/         {mov di,bx}
          $EB/$F0/         {jmp L1}
          $89/$1E/cp/      {L2: mov [cp],bx}
          $8C/$06/cp+2/    {    mov [cp+2],es}
          $26/             {es:}
          $C6/$47/$06/$02/ {mov byte ptr [bx+6],2}
          $26/             {es:}
          $8B/$6F/$04);    {mov bp,[bx+4]}

   END
   ELSE
   BEGIN
      writeln('Cannot wait only single task running');
      halt;
   END;

END;
```

End Listings

C CHEST

Listing 1 -- fdump.c, Printed 10/29/1987

```
#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>

/*  Binary file dump utility. Usage is:
 *    fdump file...    Dump all files listed on command line
 *    fdump +N file    Start N bytes from start of file
 *    fdump -N file    Start N bytes from end of file.
 */

extern long     lseek( int, long, int );

/*---------------------------------------------------------*/

#define BUFSIZE 16
#define CR      0x0d
#define LF      0x0a
#define C_CR    0x11    /* Print left arrow for CR      */
#define C_LF    0x19    /* Print down arrow for LF      */

#define isprinting(c) (' ' <= (c)  &&  (c) <= '')

static   long   Bytenum = 0L;

/*---------------------------------------------------------*/

main(argc, argv)
int     argc;
char    **argv;
{
    register int fd;
    long         offset = 0L;

    if( argc == 1 )
        usage();

    ++argv;
    --argc;

    if( **argv == '-' || **argv == '+' )
    {
        if( !isdigit( argv[0][1] ) )
            usage();

        offset = atoi( argv[0] );

        ++argv;
        --argc;
    }

    for(; --argc >= 0 ; ++argv )
    {
        if( (fd = open( *argv, O_RDONLY | O_BINARY)) == -1 )
            perror( *argv );
        else
        {
            if( offset > 0 )
                    Bytenum = lseek( fd, offset, SEEK_SET );

            else if( offset < 0 )
                    Bytenum = lseek( fd, offset, SEEK_END );

            dofile( fd );
            close ( fd );
        }
    }

    exit(0);
}

/*---------------------------------------------------------*/

usage()
{
    fprintf(stderr, "Usage fdump [+|- N] file...\n");
    fprintf(stderr, "+N   Start dump at byte N\n");
    fprintf(stderr, "-N   Start dump N bytes from EOF\n");
    exit( 1 );
}

/*---------------------------------------------------------*/

dofile( fd )
int     fd;
{
        static   char   buf[ BUFSIZE ];
        register char   nbytes;

        while( (nbytes = read(fd, buf, BUFSIZE)) > 0 )
                doline( buf, nbytes );
}

/*---------------------------------------------------------*/

doline( buf, nbytes )
char    *buf;
{
    static   char       hex[] = "0123456789abcdef" ;
    register int        i;
    register char       *p;

    printf("%06lx: ", Bytenum );
    Bytenum += nbytes;

    for( i = 1, p = buf ; i <= BUFSIZE ; i++ , p++ )
    {
            if( i <= nbytes )
            {
                    putchar( hex[ (*p >> 4) & 0xf ] );
                    putchar( hex[  *p       & 0xf ] );
            }
            else
            {
                    putchar( ' ' );
                    putchar( ' ' );
            }

            putchar( ' ' );
    }

    putchar(' ');

    for( i = 1, p = buf ; i <= nbytes ; i++ , p++ )
    {
            if( *p == CR )
                    putchar( C_CR );

            else if( *p == LF )
                    putchar( C_LF );
            else
                    putchar( isprinting(*p) ? *p : '.' );
    }

    putchar('\n');
    putchar('\r');
}
```

End Listing

Listing One *(Text begins on page 100.)*

```
Unit criterr;

 { Critical error handler, Turbo Pascal Release 4.0 }

Interface
Uses dos, crt;

{ EXTERNALLY VISIBLE PORTION }

   { The following are for saving and restoring the screen,      }
   { which is assumed to be in text mode and display page 0      }

Const bell = #7;

Type scrnPtr = ^scrnBuffer;
     scrnBuffer = array [1..4096] of byte;

Var  display, saveNode  : scrnPtr;              { display buffer }

   { The following are global variables available to the using   }
   { program to find out if an error occurred and, if so, what   }
   { it was. The program can then take appropriate action.       }

     criticalErrorOccurred : boolean;
     criticalErrorCode     : integer;
     criticalErrorDrive    : integer;
     criticalActionCode    : char;

   { The only externally visible routine installs the critical   }
   { error handler in Int 24h, replacing the DOS default.        }

Procedure InstallCEH;
Implementation
{ ------------------------------------------------------------- }

   { Following is a general-purpose critical error handler }

{$F+}
Procedure CEHandler (
      Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP : word);
Interrupt;
Var  AH, AL    : byte;
     row, col  : integer;
     action    : char;
{ -------------------------- }
  { Local functions }

  { giveReason lists reason for critical error by decoding the   }
  { low byte of the DI register. Called by procs DiskError and   }
  { CharDeviceError. Writes to screen.                           }

  Procedure GiveReason (error : byte);
  Begin
    Case error of
      $00: Writeln ('Write protect');
      $01: Writeln ('Unknown unit');
      $02: Writeln ('Drive not ready');
      $03: Writeln ('Unknown command');
      $04: Writeln ('CRC data error');
      $05: Writeln ('Bad request structure length');
      $06: Writeln ('Seek error');
      $07: Writeln ('Unknown media type');
      $08: Writeln ('Sector not found');
      $0A: Writeln ('Write fault');
      $0B: Writeln ('Read fault');
      $0C: Writeln ('General failure');
      $0D: Writeln ('Bad file allocation table');
      else Writeln ('Unknown');
    End;
  End;
{ -------------------------- }

  { DiskError is dispatched when H/O bit of AH is 0 }

  Function DiskError : word;

  Var   area, why : byte;

  Begin
    Writeln;
    CriticalErrorDrive := AL;
    Writeln ('Disk error on drive ', char (AL + 65));
    Area := (AH and 6) shr 1;                  { get AH bits 1-2 }
    Case area of
      0: Writeln ('Error in DOS communications area');
      2: Writeln ('Error in disk directory');
      3: Writeln ('Error in files area');
    End;
    Why := lo (DI);
    Write ('Type of error: ');
    GiveReason (why);
    DiskError := why;                          { error return code }
  End;
{ -------------------------- }

  { NonDiskError is dispatched when H/O bit of AH is 1. }
  { Usually triggered by a printer problem or bad FAT.  }

  Function NonDiskError : word;

  Var  why         : byte;
       deviceAttr  : ^word;
       deviceName  : ^char;
       ch          : shortInt;

  Begin
    DeviceAttr := ptr (BP, SI+4);   { point to device attr word }
    If (deviceAttr^ and $8000) <> 0 then    { if bit 15 is on.. }
      Begin
        Writeln ('Character device error');
        Write ('Failing device is ');
        ch := 0;
        Repeat
          deviceName := ptr (BP, SI + $0A + ch);
          Write (deviceName^);
          inc (ch);
        Until (deviceName^ = chr (0)) or (ch > 7);
        Writeln;
      End
    Else                                       { assume bad FAT }
      Begin
        Writeln ('Disk error has occurred');
        Write   ('Probable cause: ');
        Why := $0D;
        GiveReason (why);
      End;
    NonDiskError := why;                       { return error code }
  End;
{ -------------------------- }

Begin   { Body of CEHandler procedure }
  CriticalErrorOccurred := TRUE;               { set global flag }
  AH := hi (AX);
  AL := lo (AX);
  Col := whereX;                   { get current cursor position }
  Row := whereY;
  New (saveNode);
  SaveNode^ := display^;                 { and save screen image }
  Write (bell);                             { beep to alert user }
  If (AH and $80) = 0 then                    { if AH bit 7 = 0 }
    CriticalErrorCode := DiskError
  Else
    CriticalErrorCode := NonDiskError;
  Repeat                  { what are we gonna do about the error? }
    Write ('Abort/Retry/Ignore? ');
    Action := upCase (readKey);
    Writeln (action);
  Until action in ['A', 'I', 'R'];
  CriticalActionCode := action;
  If action = 'I' then begin  { pretend the error didn't happen }
    CriticalErrorOccurred := FALSE;
    CriticalErrorCode     := 0;
    CriticalErrorDrive    := $FF;
    CriticalActionCode    := ' ';
  End;
  Display^ := saveNode^;                  { restore screen image }
  Dispose (saveNode);
  Gotoxy (col, row);                   { restore cursor position }
  AX := 0;                          { tell DOS to ignore the error }
End;
{$F-}
{ ------------------------------------------------------------- }

   { Externally visible: installs the error handler.             }
   { NOTE: Program termination automatically reinstalls the      }
   { default handler in the vector table.                        }

Procedure InstallCEH;

Var   videoMode : byte absolute $0040 : $0049;

Begin
  SetIntVec ($24, @CEHandler);               { install in int 24h }
  CriticalErrorOccurred := FALSE;                  { set globals }
  CriticalErrorCode     := 0;
  CriticalErrorDrive    := $FF;
  CriticalActionCode    := ' ';
  If videoMode = 7 then
    Display := ptr ($B000, $0000)          { set display address }
  Else
    Display := ptr ($B800, $0000);
End;

End.
```

End Listing One

Listing Two

```
Program cerrtest;

    { Test critical error handler }

Uses  crt, dos, criterr;

Var testFile   : text;
    n, ignored : integer;

Begin
  {$I-}
  ClrScr;
  InstallCEH;
  For n := 1 to 10 do
    Writeln ('This is output line ', n);
  Assign (testFile, 'A:TEST.FIL');

  Repeat
    Rewrite (testFile);
    Ignored := IOResult;            { clear system error status }
  Until criticalActionCode <> 'R';

  Writeln ('After disk attempt, criticalErrorOccurred = ',
           criticalErrorOccurred);
  Writeln (' and criticalErrorCode = ', criticalErrorCode);
  Writeln (' and criticalErrorDrive = ', criticalErrorDrive);
  Writeln (' and criticalActionCode = ', criticalActionCode);
  {$I+}
End.
```

End Listings

THE FORTH COLUMN

Listing One *(Text begins on page 108.)*

```
****** Listing 1 for TRACY, Feb '88 ******

( Stream data and text read and write )
These utilities read and write streams of data and text from
standard or BLOCKed files.

Text lines are read into the user buffer until either the buffer
is full, or the file is empty, or an #EOF or #EOL is read.
The terminating #EOF or #EOL , if present, is not read into the
buffer.  #LF ( linefeeds) are read but ignored.

Output files are assumed to be extensible.

For your convenience, the Standard Prelude and DDJ Controlled
Reference Word Set are duplicated in this file.


( LOAD screen for DDJ Standard Prelude and String Extension)
( MJT  Nov 22 1987 for DDJ February 1987)

(     2 LOAD ( Standard prelude)
      3 LOAD ( Augmented interpretation)
   4  5 THRU ( Controlled words)
   6  9 THRU ( Strings)
  10 13 THRU ( General file support)
\    14 LOAD ( Read and write data files)
  15 16 THRU ( Read and write BLOCKed data files)
     17 LOAD ( Read text file, no   #EOL)
\    18 LOAD ( Read text file, with #EOL)
     19 LOAD ( Write text file)
  20 22 THRU ( Some examples)


( FORTH-83 functions-- typical definitions)
( Adjust these words for your Forth.    See DDJ Oct 1987.)
( Note: functions already provided need not be redefined.)
: RECURSE   [COMPILE] MYSELF ;  IMMEDIATE
: INTERPRET   INTERPRET ;

: I> ( - 'data)   COMPILE R> ;  IMMEDIATE
: >I ( - 'data)   COMPILE >R ;  IMMEDIATE

( Used for alignment: )
: ALIGN    ( HERE 1 AND ALLOT) ;
: REALIGN  ( a - a' )  ( DUP 1 AND +) ;

2 CONSTANT CELL   : CELL+   2+ ;    : CELLS   2* ;

: UNDO   I> R> R> 2DROP >I ;  \ Undoes a DO-- LOOP.
( Required definitions - used to support further compilation)

: THRU ( n n2)   1+ SWAP DO  I LOAD  LOOP ;
\ LOADS screens n through n2.

: \   >IN @ 64 + -64 AND >IN ! ;  IMMEDIATE
\ comment to end of line.  For use in screens only.

: \\   1024 >IN ! ;  IMMEDIATE
\ stops interpreting or compiling screen immediately.

: \IF ( f )   0= IF  [COMPILE] \  THEN ;  IMMEDIATE
\ conditional interpretation or compilation.

: NEED ( - f)   32 ( ie blank) WORD FIND  SWAP DROP  0= ;
\ true if the following word is in the search order.
\ FORTH-83 Controlled Words

NEED  2* \IF  :  2*    DUP  + ;
NEED D2* \IF  : D2*   2DUP D+ ;

NEED HEX \IF  : HEX   16 BASE ! ;
NEED  C, \IF  : C, ( n )   HERE 1 ALLOT C! ;

NEED BL \IF  32 CONSTANT BL

NEED ERASE \IF  : ERASE ( a n)   00 FILL ;
NEED BLANK \IF  : BLANK ( a n)   BL FILL ;

NEED .R \IF  : .R ( n width)   >R DUP 0< R> D.R ;


\ DDJ Forth Column Controlled Words
NEED 2>R
\IF : 2>R   COMPILE SWAP COMPILE >R COMPILE >R ;  IMMEDIATE
NEED 2R>
\IF : 2R>   COMPILE R> COMPILE R> COMPILE SWAP ;  IMMEDIATE
NEED @EXECUTE \IF  : @EXECUTE   @ EXECUTE ;

NEED AGAIN
\IF  : AGAIN   0 [COMPILE] LITERAL [COMPILE] UNTIL ;  IMMEDIATE
NEED DLITERAL
DUP \IF  : DLITERAL  SWAP [COMPILE] LITERAL [COMPILE] LITERAL ;
    \IF   IMMEDIATE

NEED S>D  \IF  : S>D  ( n - d)    DUP 0< ;
NEED WITHIN   \IF  : WITHIN ( n n2 n3 - f)  OVER - >R - R> U< ;
NEED TRUE \IF  -1 CONSTANT TRUE
\ String operators    See  DDJ  December 1987
\ Only  /STRING  and  EVAL  are used in this application.

: /STRING ( a n n2 - a+n2 n-n2)   ROT OVER +  ROT ROT - ;
\ truncates leftmost n chars of string.  n may be negative.

: EVAL ( a n )
\ evaluates ("text interprets") a string.
   DUP >R  TIB SWAP CMOVE  R@ #TIB !
   0 >IN ! 0 BLK !  INTERPRET  R> >IN ! ;



\\ String operators from  STRINGS.ARC  are summarized here:

ASCII ( - c)        \ returns value of following character.
CTO"" ( c - a 1)    \ converts character to string.

SKIP ( a 1 c - a2 12)
\ returns shorter string from first position unequal to byte.
SCAN ( a 1 byte - a2 12)
\ returns shorter string from first position equal to byte.

" ( - a n)      \ STATE-smart string literal.



\\ String operators from  STRINGS.ARC  continue here:
VAL? ( a n - d 2 , n2 1 , 0)
\ string to number conversion primitive.  True if d is valid.
\ Returns d if number contains ",-./:"  and sets DPL = 0
\ Returns n if no punctuation present   and sets DPL = 0<

VAL ( a n - d f)
\ converts string to double number.  True if number is valid.
\ If number contains ",-./:" then sets DPL = 0
\ If no punctuation present  then sets DPL = 0<

-TEXT ( a n a2 - -1 , 0 , 1)
\ returns -1 if string a n < a2 n , 0 if equal, and 1 if >.

COMPARE  ( a n a2 n2 - -1 , 0 , 1)
\ returns -1 if a n < a2 n2 , 0 if equal, and 1 if >.
\ The corrected version of  MATCH

: MATCH ( a n a2 n2 - ???? 0 , offset -1)
\ returns the position of string a2 n2 in (a n).
\ Offset is zero if ( a n ) is found in first char position.
\ Returns false with invalid offset if ( a n ) isn't in a2 n2.
   DUP 0= IF  2DROP 2DROP   0 TRUE EXIT  THEN
   2SWAP  2 PICK OVER SWAP -
   DUP 0< IF  2DROP 2DROP   0 EXIT  THEN
   0   ( index ) SWAP 1+ 0
   DO  ( index ) >R
     2OVER 2OVER DROP  -TEXT 0=  ( equal? )
     IF  2DROP 2DROP  R> TRUE  UNDO EXIT  THEN
     1 /STRING   R> 1+
   LOOP  2DROP 2DROP  0 ;

\ Data stream general support
1024 CONSTANT 1K

: UMIN ( u u2 - u3)   2DUP U< 0= IF  SWAP  THEN  DROP ;

\ Adjust these constants for your system:
```

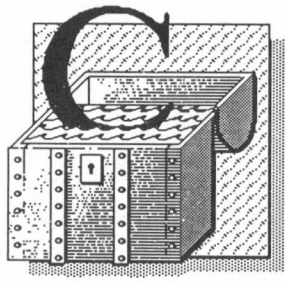
```
10 CONSTANT #LF     \ linefeed character.
13 CONSTANT #EOL    \ end-of-line character.
26 CONSTANT #EOF    \ end of file character (control-Z).

\ Adjust end-of-line and end-of-file sequence for your system:
CREATE ENDLINE   2 ( count) C,  #EOL C, #LF C,
CREATE ENDFILE   1 ( count) C,  #EOF C,



\ File size and position
\ Example of some of the structure of a file control block:

\ VARIABLE FCB   HERE FCB !  5 CELLS ALLOT  ( Containing: )
\    1 cell   current file handle-- ie selects current file.
\    2 cells  current file size in bytes (double number).
\    2 cells  current file position      (double number).

\\ You can implement CAPACITY and POSITION as 2VARIABLES.
\\ You must initialize CAPACITY to the size of your file.

2VARIABLE POSITION
2VARIABLE CAPACITY  ( eg  DSIZE CAPACITY 2! )



\ Set and reset file position
\ Given POSITION you can control the position of file access:

: MARKDATA ( - d)   POSITION 2@ ;
\ determines the position of the current file.

: SEEKDATA ( d )    POSITION 2! ;
\ changes the position of the current file.









\ Extend the file
\ If your Forth or operating system requires explicit extension,
\ supply an appropriate definition for  EXTEND .
\ Otherwise, use  : EXTEND ( d ) COMPILE 2DROP ;  IMMEDIATE

: EXTEND ( d ) COMPILE 2DROP ;  IMMEDIATE

\\
: EXTEND ( d )
\ properly extends current file by  d  bytes.
\ This example converts  d  to blocks and calls a MORE function.
   1K UM/MOD  SWAP IF  1+  THEN ( # of blocks to extend ) MORE ;



\ Read and write data files directly

: GETDATA ( a n - n2)   ;
\ reads n bytes of data from input file into address, n < 64K
\ Returns n2 bytes not read ( ie beyond end of file ).
\ Implement as a system call using CAPACITY and POSITION

: PUTDATA ( a n)   ;
\ writes n bytes of data to output file from address, n < 64K
\ Implement as a system call using CAPACITY POSITION and EXTEND



\ Read BLOCKed file as data file

: GETDATA ( a n - n2)
\ reads n bytes of data from input file into address, n < 64K
\ Returns n2 bytes not read ( ie beyond end of file ).
   ( calculate # of bytes to move < 64K : )  POSITION 2@
   BEGIN  2 PICK ( n ) DUP
     IF ( n ) >R  2DUP 1K UM/MOD  SWAP DROP  1+  1K UM*
        CAPACITY 2@ DMIN  2OVER D- 0= NOT OR  R> UMIN
     THEN  ?DUP
   WHILE  >R  2DUP 1K UM/MOD  BLOCK +  4 PICK R@ CMOVE
     R@ 0 D+  2SWAP  R> /STRING  2SWAP
   REPEAT  POSITION 2!  SWAP DROP ;



\ Write BLOCKed file as data file

: PUTDATA ( a n)
\ writes n bytes of data to output file from address, n < 64K
   ( extend the file as needed : )
   DUP 0  POSITION 2@ D+  CAPACITY 2@  2SWAP D-  DUP 0<
   IF  2DUP EXTEND  2OVER CAPACITY 2!  THEN  2DROP
   ( calculate # of bytes to move < 64K : )  POSITION 2@
   BEGIN  2 PICK ( n ) DUP
     IF ( n ) >R  2DUP 1K UM/MOD  SWAP DROP  1+  1K UM*
        CAPACITY 2@ DMIN  2OVER D- 0= NOT OR  R> UMIN
     THEN  ?DUP
   WHILE  >R  2DUP 1K UM/MOD  BLOCK +  4 PICK SWAP R@ CMOVE
     R@ 0 D+  2SWAP  R> /STRING  2SWAP   UPDATE
   REPEAT  POSITION 2!  2DROP ;

\ Read text file with #EOF

: GETTEXT ( a n - n2 f)
\ reads n bytes of text from input file into address, n < 64K
```

(continued on next page)

```
\ Returns n2 bytes not read ( ie end-of-line or beyond file)
\ Returns true if #EOL terminates line; false otherwise.
   POSITION 2@  CAPACITY 2@  2OVER D- 0= NOT OR ( limit to 64K)
   3 PICK UMIN  ?DUP 0= IF  2DROP SWAP DROP  0 EXIT  THEN  0
   DO  2DUP 1 0 D+  2SWAP 1K UM/MOD  BLOCK + C@ ( read a char )
      DUP #EOL = OVER #EOF = OR
      IF  >R  POSITION 2!  SWAP DROP  R> #EOL = UNDO EXIT  THEN
      DUP #LF -   ( a n dpos ch f )
      IF  >R  2SWAP  R@ 2 PICK C!  1 /STRING  2SWAP  R> THEN
      DROP
   LOOP  POSITION 2!  SWAP DROP  0 ;

\ Read text file without #EOF

: GETTEXT ( a n - n2 f)
\ reads n bytes of text from input file into address, n < 64K
\ Returns n2 bytes not read ( ie end-of-line or beyond file)
\ Returns true if #EOL terminates line; false otherwise.
   POSITION 2@  CAPACITY 2@  2OVER D- 0= NOT OR ( limit to 64K)
   3 PICK UMIN  ?DUP 0= IF  2DROP SWAP DROP  0 EXIT  THEN  0
   DO  2DUP 1 0 D+  2SWAP 1K UM/MOD  BLOCK + C@ ( read a char )
      DUP #EOL =
      IF  >R  POSITION 2!  SWAP DROP  R> #EOL = UNDO EXIT  THEN
      DUP #LF -   ( a n dpos ch f )
      IF  >R  2SWAP  R@ 2 PICK C!  1 /STRING  2SWAP  R> THEN
      DROP
   LOOP  POSITION 2!  SWAP DROP  0 ;

\ Read and write lines of text

: GETLINE ( a n - a n2 f)
\ reads n bytes of text from input file into address, n < 64K
\ n2 bytes are actually read; this is the opposite of GETTEXT
\ Returns true if #EOL terminates line; false otherwise.
   2DUP GETTEXT >R  -  DUP 0= 0=  R> OR ;

: PUTLINE ( a n )   PUTDATA  ENDLINE COUNT PUTDATA ;
\ writes n bytes of data to output file from address, n < 64K
\ Text stream examples

: TYPE-FILE   \ reads and prints the input text file.
\ Assumes zero-length string TYPEs nothing.
   SWITCH ( to input file saving currently active file)
   BEGIN  PAD 80  GETLINE ( n2 f)
   WHILE  CR TYPE  REPEAT  2DROP
   SWITCH ( back to current file) ;

: COPY-FILE
\ copies the input text file to the output text file.
\ Save and restore current file as needed.
   BEGIN  SWITCH ( to input  file)  PAD 80 GETLINE
          SWITCH ( to output file)
   WHILE  PUTLINE  REPEAT  2DROP  ENDFILE COUNT PUTDATA ;

\ Text stream examples
: BLOCK-TO-TEXT
\ copies the input BLOCK file to the output text file.
\ Save and restore current file as needed.
   BEGIN  SWITCH ( to input  file)  PAD 64 GETLINE
          SWITCH ( to output file)
   WHILE  -TRAILING  PUTLINE
   REPEAT  2DROP  ENDFILE COUNT PUTDATA ;

: TEXT-TO-BLOCK   0 ( previous line length )
\ copies the input text file to the output BLOCK file.
   BEGIN  SWITCH ( to input  file)
          PAD  64  2DUP BLANK  GETLINE  ROT ( a ) DROP
          SWITCH ( to output file)
   WHILE  DUP 0= ROT 64 = AND NOT IF  PAD 64 PUTDATA  THEN
   REPEAT  2DROP ;
\ Text stream examples

: EVAL-FILE   \ reads and interprets the input text file.
\ Assumes zero-length interpreted string does nothing.
   SWITCH ( to input file saving currently active file)
   BEGIN  PAD 80  GETLINE ( n2 f)
   WHILE  EVAL  REPEAT  2DROP
   SWITCH ( back to current file) ;
```

End Listing

Hiding Configuration Information



Storing a program's configuration information—variables that can be changed by users but that must retain their values between successive program invocations—is a common problem. An easy solution is to use a configuration file that's read in by the program when it boots.

Configuration files have problems, though. The first one is finding the file. Many programs require the configuration file to be in the current directory. If you're going to execute the program in more than one directory, you need a configuration file in each of these directories. Clearly this behavior isn't really acceptable. Not only do you start filling up your disk with unnecessary files but also the files can get out of sync with each other—if you make a change to one, you have to make a change to all of them.

There are other solutions, however. First, you can search for the file along the *PATH*. Example 1, page 95, shows a search-for-file subroutine that can be used for this purpose. It's passed two strings—the first containing a file name, and the second the name of an environment that holds the search path (a semicolon-delimited list of directories). The *search( )* subroutine looks for the file, first in the current directory, and then in all the directories listed in the given environment string. If the file is found, *search( )* returns a pointer to the full path name; otherwise it returns *NULL*.

by Allen Holub

The *access( )* subroutine that *search( )* uses is a Unix function that looks at the permission mask associated with a file. You can use it to test for read permission, write permission, and so forth. Here, I'm just using it to test for existence. If you don't have an *access( )* function, you can do the same thing by trying to *open( )* the file for read and looking to see if *open( )* returned an error (if it did, the file didn't exist).

The *strpbrk (char *src, char *pat)* function searches the *src* string for any of the characters in the *pat* string and returns a pointer to that character if found (*NULL* if not). The *strtok (char *src, char *delim)* function extracts a series of tokens from the *src* string. Tokens are delimited by any of the characters in the *delim* string. The first time the subroutine is called, it returns the first token from the string. In subsequent calls, the first argument is set to NULL, and it returns subsequent tokens from the original string. It returns NULL when there are no more tokens.

The *getenv( )* function returns the contents of the indicated environment. This string has to be copied to *pbuf[ ]* because it's modified by the subsequent *strtok( )* calls. *Strpbrk( ), strtok( ),* and *getenv( )* are all ANSI functions, so they should be in your compiler's library.

Another alternative to searching along the *PATH* is provided by some compilers, such as Microsoft's. These compilers provide the full path name of the executable file in *argv[0]*. You can then require the configuration file to be in the directory as the executable file.

A third (and, I think, the best) alternative is to dispense with the configuration file entirely and to incorporate the configuration information in the .EXE file itself. This way, you don't clutter up the disk with needless files whose immediate purpose is not obvious. To use the .EXE file, you have to find it on the disk, using either of the methods discussed earlier. You also have to declare a structure in your program that will contain the modifiable options. At very least this structure must contain a signature field and a checksum. The signature contains an arbitrary, but unchanging, string that you can look at to see if the options are valid.

A stripped-down options structure (called *Opts*) is shown in Example 2, page 96. The Microsoft compiler correctly evaluates *sizeof(DEF_SIG)* as the number of characters in the string (including the \0). I can't vouch for other compilers, though. Because the length is used in a declaration, you can use a *strlen( )* call to compute it (because it's executed at run time, not compile time). Consequently, if your compiler's *sizeof* doesn't work correctly, you'll have to count the characters in the signature string to declare the array.

Options are fetched from the .EXE file with a call to *get_opts(argv[0])* at the head of my *main( )* subroutine. Here, *argv[0]* holds the full path name of the .EXE file. If this is the first time that the program is executed, *get_opts( )* creates (and initializes) the options area in the file. *Get_opts( )* is shown in Example 3, page 96.

Options are stored at the end of the .EXE file, following any executable code. The .EXE file is opened for binary-mode read on line 14. I then seek to what ought to be the beginning of the options area on line 20. That is, the options are at the end of the file, so I seek to end of file less the size of the *Opt* structure. The structure is loaded on line 22. Now I look at the signature (on line 25). If it doesn't match, I assume that this is the first time that the program has been run, in which

(continued from page 92)

case the options won't exist yet. So, I initialize the *Opt* structure myself on lines 27–38.

A checksum for the structure is computed on line 39–42. The checksum is the negative sum of the other bytes in the structure. That is, if you add up the contents of the entire area occupied by the structure (including the checksum), you'll get 0. The checksum has two purposes: obviously, it can be used for checking the validity of the data in the structure itself; it also keeps DOS happy because every .EXE file also has a checksum. The structure's local checksum cancels any effect that the rest of the structure's contents would have on the .EXE file checksum. That is, because the sum of all the bytes in the structure is 0, the presence or modification of the structure won't change the file's checksum.

I seek to end of file and write out the initialized options area on lines 43–48. I *#ifndef*ed out the actual *write( )* call when I'm debugging because CodeView, as it also puts stuff at the end of the .EXE file, gets confused if it can't find its own information there. That is, it thinks that there's no debugging information if I add my own data to the end of the file.

If any options were changed during the run, the options area in the .EXE file is updated when the program terminates with a call to *put_opts(argv[0])*, where again *argv[0]* holds the full path name of the .EXE file. *Put_opts( )* is shown in Example 4, page 96. It opens the .EXE file and then recomputes the checksum to reflect the changes made during the run. The routine then seeks in the file to the beginning of the options space (which must exist) and writes out the modified structure. Again, I *#ifdef* out the actual write if I'm debugging to keep CodeView happy.

Nifty Stuff: awk

awk, in addition to being a flightless, seagoing bird and the sound made by that bird, is one of the more useful tools in the Unix toolbox. Many (perhaps most) programs do

```
#include <stdio.h>

extern char *strpbrk(),          /* Library routines */
            *strtok(),
            *getenv();

static  char    *search( file, env_name )
char    *file, *env_name ;
{
    /* Search for file by looking in the directories
     * listed in the env_name environment.
     * Return a pointer to the full path name if you
     * find it (NULL if you don't). The returned string
     * is transient; it will be modified by the next

     * call to search().
     */

    static   char       pathname[80];
    char                pbuf[129];
    char                *p ;

    strcpy( pathname, file );

    if( access( pathname, 0 ) != -1 )
        return pathname;

    /* The file doesn't exist in the current directory.
     * If a specific path was requested (ie. if
     * file contains the characters \ or /) or if
     * the PATH environment isn't set, return a NULL;
     * else search for the file along the path.
     */

    if( strpbrk(file,"\\/")  ||  !(p = getenv(env_name)) )
        return NULL;

    strncpy( pbuf, p, 129 );

    if( p = strtok( pbuf, ";" ) )
    {
        do
        {
            sprintf(pathname, "%0.50s\\%0.20s",
                              p, file );

            if( access( pathname, 0 ) >= 0 )
                return pathname;
        }
        while( p = strtok( NULL, ";") );
    }
     return NULL;
}
```

Example 1: *Search( )*

(continued from page 95)

nothing but manipulate or verify data. Filter programs, such as grep, sed, pr, and so forth, just output a shuffled around version of an input file. Other programs, such as database report generators, shuffle around a database and output parts of it in an ASCII representation. Though none of these programs are particularly hard to write, it's a nuisance to write a hoard of special-purpose programs in a language such as C—especially if you're going to use that program once and throw it away.

awk provides a solution to this problem. It is essentially a dialect of C that's optimized for text processing—a general-purpose tool from which other tools can be built. It supports all the C operators and control-flow statements (even recursion), though the operators have been extended to work with strings. For example:

if("aardvark" < "zebra")

evaluates to true.

awk programs all take the form *expression* { *action* }, where the expression tells awk when to apply the action. That is, the action can be applied on every line, on a range of lines, a group of lines delimited by a specific string, on every line that contains a match for a regular expression, on every line that has a specific string or number in a specific field, and so forth. The action can be a simple *printf( )* statement (*printf( )* is an awk primitive) or a complex program that does elaborate database manipulation. Most of the familiar C constructs are available.

As an example, the following simple awk program numbers all

```
#include <stdio.h>
#include <fcntl.h>

#define DEF_SIG  "(C) 1987, Allen I. Holub. All rights reserved."

struct options
{
    char    signature[ sizeof(DEF_SIG) ];
    int     chksum;
    /*************************************************/
    /* This structure also contains fields for every */
    /* option that the user can change at run time.  */
    /*************************************************/
}
Opt ;
```

Example 2: *Options header*

```
get_opts( name )
char    *name;
{
    /* Various statistics (such as last time the program was
     * run) are stored in a buffer at the end of the
     * executable file. These stats are all stored in a
     * "options" structure. This subroutine creates the data
     *  area if it doesn't exist and initializes Opt as
     *  appropriate. /

    int  fd;
    int  i, *p;          /* Used to create checksum */

    if( (fd = open( name , O_RDWR | O_BINARY )) == -1 )
    {
        perror ( name );
        exit   ( 1    );
     }

    lseek ( fd, 0L - sizeof(Opt), SEEK_END      );

    if( read( fd, (char *) &Opt, sizeof(Opt) ) != sizeof(Opt))
        ferr( "Internal error: Can't read options\n" );

    if( strcmp( DEF_SIG, Opt.signature ) != 0 )
    {
        memset( &Opt, 0 , sizeof(Opt)); /* The memset is for
                                         * debugging.(it shows
                                         * us that the record.
                                         * has been written
                                         correctly./

        strcpy ( Opt.signature, DEF_SIG  );

        /*******************************************************/
        /* Initialize other fields in the Opt structure here */
        /*******************************************************/

        Opt.chksum = 0 ;
        for( p = (int *)(&Opt), i = sizeof(Opt)/2; --i >= 0; )
                Opt.chksum -= *p++;

        lseek ( fd, 0L, SEEK_END );

#       ifndef DEBUG
            if( write( fd, (char *) &Opt, sizeof(Opt))
                                        != sizeof(Opt) )
                ferr("Internal error: Can't initialize
                                              options\n");
#       endif
    }

    close( fd );
}
```

Example 3: *Get_opts( )*

```
put_opts( name )
char    *name;
{
    /* Update the options buffer (which better exist).
     * If the options buffer doesn't exist, this
     * subroutine will destroy the end of the file.
     */

    int  fd, *p, i, checksum ;

    if( (fd = open( name , O_RDWR | O_BINARY )) == -1 )
    {
        perror( name );
        exit(1);
    }

    /* Recompute the checksum */

    checksum = 0 ;
    for( p = (int *)(&Opt), i = sizeof(Opt)/2; --i >= 0; )
            checksum -= *p++;


    if( checksum != Opt.chksum )
        printf( "Options have changed" );

    Opt.chksum = checksum;

#   ifndef DEBUG
        lseek ( fd, 0L - sizeof(Opt), SEEK_END );
        if( write(fd, (char *) &Opt, sizeof(Opt))
                                    != sizeof(Opt) )
            ferr( "Can't do final options update\n" );
#   else
        printf("\n***** No I'm not" ****** );
#   endif

    close( fd );
}
```

Example 4: *Put_opts( )*

```
BEGIN { pageno = 0; }

{
    if( (NR % 55) == 0 || pageno == 0 )
    {
        if( pageno )            # not the first page
            printf("\f");

        printf("%s, page %d\n", FILENAME, ++pageno );
        printf("------------------------------\n\n");
    }
    printf("%3d: %s\n", NR, $0)
}

END { printf("\f"); }
```

Example 5: *Listing.awk, an awk program to create listings*

the lines in the input file and sends the result to standard output:

```
awk '{printf("%3d: %s/', NR, $0)}'
input
```

This one-line program is made more useful in the awk program in listing.awk, Example 5, page 96. When executed with:

```
awk –f listing.awk input
```

a file called input is read and then printed with all the lines numbered. A header giving the file name and page number is printed at the top of every page, and a form feed is printed at the bottom. Looking at the program itself, the *BEGIN* action is done before any input is processed. Here it sets *pageno* to 0. Variables are declared implicitly by using them. The *END* statement is executed at the end of the input. Here it prints a form feed. *NR* and *FILENAME* are predefined variables that hold the current line number and input file name. Because there's no specific pattern or other line selector to the left of the *action*, it's performed on every line. The # delimits a comment.

To go to the other extreme, Example 6, page 98 (extracted from the book discussed later), holds a version of the Unix make utility, written entirely in awk. I've included it here primarily so that you can see the power of the awk programming language. *Getline* is a built-in function that reads a line of input (in this case from *makefile*). Similarly, *sub* substitutes matches of the regular expression given as the left argument with the pattern given as the right argument. *Print*, *printf*, *system*, and *close* are also built-in functions that work as you would expect from their names. *$1*, *$2*, and so on are the fields on the line (fields are space- or tab-delimited by default), and *$0* is the whole line. The *~ (tilde)* operator is the matches operator, so *$0 * /~[A–Za–z]/* checks to see if the first character on the line is a letter.

To my surprise and delight, awk is now available to us masses. *The AWK Programming Language* by Alfred Aho, Brian Kernighan, and Peter Weinberger is an excellent introduction to the language itself, and an executable version called MKS AWK is available for the IBM PC and clones from Mortice Kern Systems (43 Bridgeport Rd. E, Waterloo, Ontario, Canada N2J 2J4) for $75.

The book is both lucid and quite readable (surprising considering the turgid prose in Aho's other books). It starts out with a short tutorial introduction to awk. The next chapter is a complete (but somewhat dry) language description, and the remainder of the book contains awk programs, some quite complex. There are chapters on general data processing; database management and report generation (an awk query language is presented); word-processing applications (for example, an index generator that works with troff); language processing (such as a graph-generating language that takes a graph description as input and outputs an actual graph); and more (for example, a few fancy sort programs, including a heap sort and a topological sort program that works like the Unix tsort, are presented).

I've only one complaint about the book: the actual code is often poorly formatted and undercommented, so some of the examples are difficult to read. *The AWK Programming Language* is much like Kernighan and Ritchie's *The C Programming Language* in this respect. Nonetheless, as in K & R, the examples are often instructive and the time spent deciphering them is well spent.

MKS AWK is a complete implementation of the awk language described in Aho's book. It's quite solid and very much like the Unix System V (Release 3.1) version (some would say too much like Unix—the ubiquitous *syntax error* is printed for virtually every typo that you make in the source file). Four versions of the program are supplied—a large model, with and without an 8087, and a small model, with and without an 8087. In addition, versions of the Unix date, glob, join, sort, and tr programs are provided along with a DOS-specific program that let's you change the switch character used by COMMAND.COM from a / to something more reasonable (thereby letting you use / in path names).

Documentation is supplied in a

5.5- × 8.5-inch saddle-stitched booklet. It is adequate for describing the awk language but is by necessity not as complete as Aho's book. I'd recommend getting both the book and the program, even though the MKS documentation contains everything you need to get started.

In all, awk is a remarkably useful tool. It was the one major Unix utility that I hadn't seen running under DOS, and it's a welcome addition to my toolchest. The Mortice Kern implementation is very good; I recommend it highly.

Books and File Dumps

Like most people, when it comes to work, I have the best of intentions but rarely manage to get enough accomplished. Consequently, I've an ever-growing stack of books to review gradually taking over what little bare space is available on the top of my desk. This month I'll look at one of these and hopefully clear up the rest of them in upcoming months.

Harbison, Samuel P.; and Steele, Guy L., Jr. *C: A Reference Manual*. 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, 1987.

Harbison and Steele's book has long been the best reference available on the C language—much better, in fact, than Appendix A of K & R. Prentice-Hall has just published a second edition that makes the book even more valuable than before. The new edition has been updated to include the various ANSI extensions (at least, the extensions as they stood in late 1986). It describes the complete C language in considerable detail—the book was originally intended to be the specification for a compiler project, so it goes into the language with the detail necessary to actually write a compiler—and it also covers all the ANSI library functions. Particularly valuable are discussions of implementation-dependent issues that are likely to affect portability.

There is one unfortunate omission from the second edition. The first edition had two formal grammars for C: one was intended to show you the language syntax; the other was a less readable but more practical grammar, such as you would submit to YACC. The second of these has been left out of the new edition.

Nonetheless, this is a very valuable book that should be in every C programmer's library. I can't recommend it too highly.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Errata:

Last month's column examples contained some errors:

delete all "\sc128\".

Example 3, line 14 should read:

```
int sam( ), dave( ), timer( ), idle( ),
                              maintask( );
```

Example 3, line 35 should read:

```
printf("%ld interrupts, %ld
            blocked\n", t_numint( ),
                              t_numblk( );
```

Example 9, line 7 should read:

```
:(xhat + = (data - xhat) / + +ki)
```

DDJ

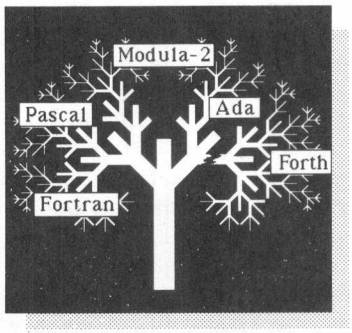
(Listing begins on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

```
#
# A make version of AWK. Taken from The AWK Programming Language
# by Aho, Kernighan, and Weinberger, p. 178.
#
BEGIN {
    while( getline <"makefile" > 0 )            # look for  $1: $2 $3
        if( $0  /^[A-Za-z]/) {
            sub(/:/,"")

             if (++names[nm = $1]] > 1)
                error(nm " is multiply defined")

            for( i = 2; i < NF; i++ )   # remember targets
                slist[nm, ++scnt[nm]] = $i

        } else if ($0  /^\t/)                   # remember cmd for current name
            cmd[nm] = cmd[nm] $0 "\n"

        else if ( NF > 0 )
            error( "illegal line in makefile: " $0 )

    ages()                                      # find initial ages

    if( ARGV[1] in names )
    {
        if (update(argv[1]) == 0 )
            print ARGV[1] " is up to date"
    }
    else
        error( ARGV[1] "is not in makefile" )
}
function ages( f, n, t )
{
    # execute ls -t (which gives a list if files sorted by time last
    # modified) and pipe the result into getline. That is, the for
    # loop process the output from the ls command, one line at a time

    for( t = 1; ("ls -t" | getline f) > 0; t++ )
        age[f] = t                      #all existing files get an age

    close("ls -t")
    for( n in names )
        if( !(n in age) )               # if n has not been created
            age[]n] = 9999              # make n really old
}

function update(n, changed, i, s )
{
    if( !(n in age  )) error( n "does not exist" )
    if( !(n in names)) return 0

    changed = 0
    visited[n] = 1

    for( i = 1; i <= scnt[t]; i++ ) {
        if( visited[s = slist[n,i]] == 0)
            update(s)
        else if( visited[s] == 1 )
            error(s "and " n "are circularly defined")

        if( age[s] <= age[n])
            changed++
     }
    visited[n] = 2
    if( changed || scnt[n] == 0 )
    {
        printf("%s", cmd[n])
        system(cmd[n])                  # execute cmd associated with n
        ages()                          # recompute all ages
        age[n] = 0                      # make n very new
        return 1
    }
    return 0
}

function error(s) { print "error: " s ; exit }
```

Example 6: *An awk implementation of make*

Writing a DOS Critical Error Handler



An application running under DOS gets ugly treatment when a hardware error occurs. The alternatives DOS provides are the infamous Ignore (at peril), Retry (and get the same error again), and Abort (losing the work done so far). To get around these nasty consequences, you can write a special kind of interrupt service routine called a critical error handler. That's what I'll do here, and in the process I'll point out some of the features that make the new Turbo Pascal, Version 4.0, a real treat for serious application developers.

The critical error handler wakes up (via interrupt *24h*) when DOS encounters any of 14 problems associated with disk and character devices. These problems run the gamut from an open drive door to media failure. The default critical error handler built into DOS displays a message explaining the problem and asks users to select from the Abort/Retry/Ignore alternatives. Because the default handler sets no state switches and returns nothing, DOS applications that don't replace it can't make any provision for critical errors. The best you can hope for is that the error will go away if the user selects Retry—fat chance.

A better solution is to take over the vector for interrupt *24h*, pointing it to your own handler, which records what went wrong and sets a flag, then returns. Your application can check the flag after each disk or printer I/O request and, if it finds

by Kent Porter

that a critical error occurred, recover or shut down gracefully; the action depends on the nature of the error.

There are two classes of critical errors—disk and nondisk. Although DOS triggers interrupt *24h* for both, they require different processing. On entry, bit 7 of register *AH* contains a 0 for disk errors and a 1 for nondisk problems. Because interpretation of the registers differs from that point on, entry processing should test this bit and branch to the appropriate routine.

Disk errors occur more often than nondisk errors do, and there are more possibilities. Consequently, DOS passes a number of items of information, especially in register *AH*. Bit 0 contains 0 if a read failed and 1 if a write failed. Bits 1 and 2 show where the error was detected, the patterns being:

00	DOS work area
01	File allocation table
10	Disk directory
11	Files area

Register *AL* indicates the drive on which the failure took place, the values being 0 for A, 1 for B, and so on. The low half of the *DI* register contains an error code (and the upper half garbage, so isolate the low byte before attempting to interpret the code). Finally, the *BP:SI* register pair points to the device driver header, although nobody cares in handling disk errors. The failure codes returned in *DI* are listed in Table 1, page 103.

The nondisk error class is a catch-all for two entirely unrelated kinds of problems. One occurs when a character device—nominally a printer, but it could be a serial port, too—signals a malfunction. The other occurs when DOS detects a discrepancy between the two in-memory copies of the file allocation table (FAT), which governs the management of disk space.

To determine which error is being reported, use the address in *BP:SI* to check the attribute word in the device driver. It's at offset 4 from *BP:SI*. If bit 15 is off, the FAT is corrupted; if it's on, you have a problem with a character device. In the latter case, you can determine which device failed by inspecting the string at offset 8 from *BP:SI*. It gives the logical device name (LPT1, COM1, and so on) of the guilty party.

The critical error handler built into DOS uses this information to produce a brief explanation of the problem. It then asks the user the infamous Abort/Retry/Ignore question and, based on the answer, passes one of three codes back to DOS in the *AL* register. A code of 0 means Ignore (that is, pretend nothing happened and restore control to the running program); 1 means Retry the operation; and 2 means Abort the program without giving it a chance to close files.

Because the purpose of writing a custom error handler is to avoid the catastrophic consequences of these choices, your handler should record information about what went wrong, set a Boolean flag (*CriticalErrorOccurred*) to *TRUE*, and always return 0 in register *AL*. That way DOS will restore control to your program, which can check the flag with the test:

```
if CriticalErrorOccurred then ...
```

and take appropriate action.

And what might that action be? Well, it depends. If the disk is unformatted (error code 8), you might execute *FORMAT* as a child process using Turbo Pascal 4.0's *EXEC* procedure. For drive not ready (code 2), you could tell the user to stick in a disk and close the door, then retry. Unrecoverable errors call for more

(continued from page 100)

drastic action, such as closing all files and terminating the program. The error handler itself only records the bad news; what you do about it is up to you.

There's a more or less ironclad rule among DOS hackers that you shouldn't attempt any I/O from inside an interrupt handler. This is because of the notorious DOS reentrancy problem. An exception is made for a critical error handler, however; it can perform console I/O functions (DOS interrupt *21h*, functions *01h* through *0Ch*). Turbo Pascal and most other high-level languages use these very functions to communicate with the user, so you can safely carry on a console dialog from within the handler.

Examining the Code

Now let's look at the real live critical error handler in Listing One (see page 84). It's written as a unit, a new feature of Turbo Pascal 4.0 that provides for separately compiled, linkable modules.

The only externally visible routine in the *criterr* unit is the *InstallCEH* procedure. It's called by the using program to stuff the address of the critical error handler into the vector for interrupt *24h*, thereby activating the handler. It also initializes the variables set by the handler and determines the location of the video buffer so that the handler can save and restore the display image without permanently corrupting it.

Incidentally, it's not necessary to restore the vector to the default DOS routine when the program ends. That's because DOS automatically does this as part of job termination processing. For that reason, the unit lacks an uninstall procedure.

The handler itself occupies the rest of the unit and is sufficiently generalized to serve as a model. This handler sets four global variables when it gets control: a Boolean flag to indicate that an error occurred, the error code, the drive if a disk, and an action code (the initials of Abort, Retry, and Ignore).

Notice the heading for *CEHandler* in the *Implementation* section of the unit. The *$F+* switch forces far calls, thus guaranteeing that the installation procedure will get a full 32-bit segment:offset pointer to set the interrupt vector. The parameters to *CEHandler* are the general registers. Because of the *Interrupt* keyword following the heading, the compiler saves the named registers on the stack at entry, making their contents available as local variables.

The body of the handler begins after the third local subroutine. It immediately sets the error flag and dissects the *AX* register into its two byte-size components. Next the handler saves the cursor position and copies the screen image onto the heap to prevent it from being corrupted. After determining the error class, it dispatches the appropriate subhandler to report the problem to the user. It then determines what the user wants to do about it (Abort/Retry/Ignore) and records it as the action code. If the user says to Ignore, it resets the four reporting variables. After restoring the screen image and cursor position, it zeros the *AX* register, telling DOS to pay no attention, and returns from the interrupt.

The *DiskError* and *NonDiskError* routines are called from the main body, depending on the error class. Both call the *GiveReason* procedure, which merely prints a diagnostic message on the screen based on the error code passed to it.

DiskError is straightforward, loading the error drive global and advising the user of the location and nature of the problem. The *NonDiskError* function is a little more complex because it deals with two distinct kinds of errors. The branch is based on the high-order bit of the device attribute. Note the *Repeat... Until* loop that outputs the device name; it can be up to eight characters long, but if it's shorter, the unused bytes are ASCII 0s. Consequently, the loop halts on a null or after the eighth character, whichever comes first.

Strung throughout the handler are instructions that set the appropriate error indicators so that the using program can sense what went haywire and decide what corrective action to take.

Testing It Out

Make sure the door to drive A is open when you run the cerrtest program in Listing Two, page 85. The program attempts to create a file on A. If it can't, the handler in the *criterr* unit gains control and reports the problem. The program then displays the error globals and quits. Because the error handler saves the screen before output, then restores it when it's finished, the error dialog simply vanishes.

The *Uses* statement at the top of Listing Two illustrates how units get linked with programs in Turbo Pascal 4.0. This program uses three units, the first two furnished with Turbo Pascal and the last the critical error handler from Listing One. Anything in the *Interface* section of a unit is accessible to the using program: constants, types, variables, and subroutines. Consequently, although *cerrtest* doesn't declare the *CriticalError* variables or define the *InstallCEH* procedure, it has access to them via the interface to *criterr*.

You might wish to make your own critical error handler less user dependent. In that case, remove all the I/O from *criterr* and simply load the reporting variables; your program can then decide what to do and how much to tell the user. By building in a custom critical error handler, you'll make your software much more bulletproof.

DI	Means
00h	Write-protected disk
01h	Invalid drive designator
02h	Drive not ready: empty or door open
03h	Unknown command: probably bad DOS call
04h	CRC data error: bad disk
05h	Invalid request structure: program error
06h	Seek error: hardware failure
07h	Unknown media type: probably bad disk
08h	Sector not found: usually unformatted disk
0Ah	Write fault
0Bh	Read fault
0Ch	General failure

Table 1: *Critical error failure codes returned in* DI

STRUCTURED PROGRAMMING
(continued from page 104)

Who Am I?

Because there's a new name on this column and it's mine, perhaps I should introduce myself. I've been hanging around computer rooms since the early 1960s doing various software and management jobs, and I was one of the micro pioneers when I got an IMSAI 8080 back in the Dark Ages, which was nine years ago. My first book was *Computers Made Really Simple*, published in 1976 and long out of print. Since then, I have written more than a dozen more, the most recent being *Stretching Turbo Pascal* (he mentioned with shameless commercialism). I've written another Pascal book as well, and I'm working on my second C book at the moment. I also write a lot of magazine articles, not only about programming but also about the application of small computers to business problems.

What I intend to do in this column is to present programming solutions—this installment being an example—as well as to review structured programming products and discuss issues relevant to software development.

But this isn't just my column, it's ours. If there's something you'd like to see here, drop me a line at *DDJ*. Or you can leave a note for KPORTER on MCI. Don't call, though, please; I don't work at the editorial offices. No promises, but I'll consider any reasonable suggestion. Right now I'm one guy in a room by himself, trying to guess what The Reader (that's you) wants. I won't know until you tell me. Let's make this into a forum that's fun and instructive for us both.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

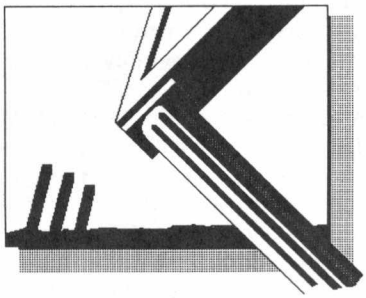
DDJ

(Listings begin on page 84.)

Vote for your favorite feature/article.
Circle Reader Service No. 2.

THE FORTH COLUMN

Update on Forth in the Industry



by Martin Tracy

This last year has been a busy one for Forth. I like to believe that the "Forth year" starts and ends with FORML (Forth Modification Laboratory) at Asilomar in Northern California over the Thanksgiving weekend. I'll give you a complete report of this year's FORML (87) in the next column. Meanwhile, here's what's happened since the last FORML (86). I apologize in advance for missing anything.

The Novix 4000 (renumbered to 4016) continued rising in popularity. Both Novix and Software Composers released Novix boards for the IBM PC. The Novix NB4100 contains the NC4016 with B and X ports wired to board-mounted connectors and 128K of program and data memory. The Software Composer PC4000 has 500K of memory on-board and can run in parallel with other PC4000 boards. Each company offers Forth development systems as well as C-to-Forth translation programs. Novix also announced the NB4300 STD-bus Novix card. (An STE-bus Novix card is available from Forth Systeme, W. Germany.)

Harris Semiconductor announced FORCE (Forth-optimized RISC computing engine), a modified NC4016 design copied into its macro cell library. It expects to offer a FORCE-based, real-time control processor (RTCP) in the first quarter of 1988.

Phil Koopman demonstrated his WISC (writable instruction set, stack-oriented computer) CPU/32, a modular hardware Forth engine sold by Mountain View Press. The team from Johns Hopkins University demonstrated its Forth engine, a 32-bit processor with cached stacks and Gbytes of address space. The team expects it to be generally available sometime in 1988.

Meanwhile, Forth continued to move into newer and more challenging hardware environments. Dr. C.H. Ting implemented a Novix-based micro-code sequencer for NCR's GAPP computer. Goddard Space Lab (Maryland) implemented Forth on its massively parallel processor, a two-dimensional array of 128 × 128 serial processors. FORTH Inc. announced a polyFORTH for the Texas Instruments TMS32020/C25 digital signal processor (DSP). Forth is currently the only high-level language running directly on this popular DSP chip.

FORTH Inc. also implemented a nicely optimized Intel 80386 polyFORTH running in native mode. The company reports that Forth benchmarks run faster on this machine than they do on the Motorola 68020. Meanwhile, Laboratory Microsystems demonstrated its new UR/FORTH running on the 80286 and 80386 under Microsoft's OS/2.

Forth vendors were busy last year applying their tools to a variety of projects. Miller Microcomputer Services put the finishing touches on MMS Forth 2.4, the MS-DOS version of Forth used to develop RapidFile, Ashton-Tate's new query-by-example database. The new book *Business Programming with RapidFile* was written by our own Leo Brodie, whose improved *Starting Forth*, second edition, was also published last year.

Creative Solutions turned its talents inward to develop new hardware technology. The result was a series of boards for Apple's Macintosh II NuBus. Three of these Hurdler boards connect the Mac II to three other buses. the STD bus, the Motorola I/O channel, and the IBM PC bus. CSI also announced the Mac II Toolkit, which adds a 68020 assembler, 6881 support, and color graphics as extensions to its popular MacFORTH.

FORTH Inc. spent a busy year working with several *Fortune* 500 companies. The company assisted in developing a package-tracking magic wand for a major mailing company and renovated the American Airlines baggage system, a polyFORTH system that has been in place for the past five years. It expanded Bell Canada's data entry system (70–80 users on one VAX VMS with no loss of response) and demonstrated a large factory heating ventilation and air-conditioning system using the ClusterFORTH distributed intelligence network.

Advance MicroMotion developed an economy Telex and EasyLink communications package for TTS. Mountain View Press brought out new MVP Forths for the Amiga, the Atari 600/800/1200, and the PDP-11. New Micros announced a hardware and software development system for the Motorola 68HC11, and Inner Access Corp. announced its development system for the Zilog SUPER8 chip.

Besides RapidFile, two other major Forth products appeared. Electronic Arts' STARFLIGHT game took several programmer-years of effort (see *Forth Dimensions*, July 1987). Frog Peak Music (P.O. Box 9911, Oakland, CA 94613) brought out the Hierarchical Music Specification Language, which lets you write your own synthesizer MIDI driver and edit waveforms, durations, and envelopes as you play.

Last year also saw the birth of a new Forth computer, sort of. The Canon Cat (*Byte*, October 1987) is Jef Raskin's first new machine since he left Apple, where he headed the original Macintosh team. The Cat has a

9-inch mono display, 3.5-inch drive, and a Motorola 68000 CPU. According to Ezra Shapiro, "If the highlighted text [on the screen] is a computer program written in either Forth or 68000 assembly language, the Cat executes it."

And of course, the ANSI CBEMA X3J14 Forth standardization effort began last year. I'll report on the second meeting of this committee in the next column. Also, there are two new Forth electronic bulletin boards: the FIG-sponsored GENie board (see *DDJ*, December 1987) and the new North Coast Forth Board (Minnesota [612] 483-6711). Now there are Forth boards north, east, and west. South Coast Forth Board anyone?

Forth in AI

Forth continued to make inroads into artificial intelligence. Henry Harris (JPL) presented papers on conceptual dependency; William Dress (Oak Ridge National Lab) presented several on neural nets. Two implementations of OPS5 and two of "fuzzy" rule production systems were developed last year. Jack Park's popular Expert 2 system evolved to Expert 5. For a general summary of Forth's recent history in AI, see "The Forth Wave in AI" by Robert Trelease in *AI Expert* (October 1987).

Part of Forth's popularity in AI is the ease in which it can be implemented on experimental computer architectures. This makes Forth particularly attractive for simulating, controlling, and testing inference engines and neural networks. Once Forth is ported to a new hardware environment, it is possible to implement additional languages, such as BASIC, LISP, and PROLOG, rapidly by writing them in Forth. Panasonic, for example, implemented BASIC this way on the original HHC (handheld computer). Charles Duff's

```
/STRING ( a n n2 - a+n2 n-n2)
\ truncates leftmost n chars of string.
\ n may be negative.

EVAL ( a n )
\ evaluates ("text interprets") a string.

ASCII ( - c)
\ returns value of following char.

CTO"" ( c - a 1)
\ converts character to string.

SKIP ( a 1 c - a2 12)
\ returns shorter string from
\ first position unequal to byte.

SCAN ( a 1 byte - a2 12)
\ returns shorter string from
\ first position equal to byte.

" ( - a )
\ STATE-smart string literal.

VAL? ( a n - d 2 , n2 1 , 0)
\ string to number conversion.
\ True if d is valid.
\ Returns d if number contains ",-./:"
\ and sets DPL = 0
\ Returns n if no punctuation present
\ and sets DPL = 0<

VAL ( a n - d f)
\ converts string to double number.
\ True if number is valid.

-TEXT ( a n a2 - -1 , 0 , 1)
\ returns -1 if string a n < a2 n ,
\ 0 if equal, and 1 if >.

COMPARE ( a n a2 n2 - -1 , 0 , 1)
\ returns -1 if a n < a2 n2 , \ 0 if equal, and 1 if >.

MATCH ( a n a2 n2 - ???? 0 , offset -1)
\ returns the position of
\ string a2 n2 in (a n).
\ Offset is zero if ( a n )
\ is found in first char position.
\ False with invalid offset
\ if ( a n ) isn't in a2 n2.
```

Example 1: *String operators*

object-oriented NEON and Actor were both written as extensions to Forth, too.

By the way, a tiny Modula-2 has just been written in Forth by S. Lohr. You can download it from the East Coast Forth Board ([703] 442-8695) or from the new GENie Forth Board as the file TM2.ARC. You will find an interesting discussion of language bootstrapping techniques in "Embeddings of Languages in Forth" by R.D. Dixon in the latest *Journal of Forth Application and Research*, vol. 4, no. 4, 1987.

This same issue contains two important articles on implementing PROLOG in Forth: "A Full PROLOG Interpreter Embedded in Forth" by Odette and Paloski and "Compiling PROLOG to Forth" by Odette. The latter article contains careful and expert notes on implementing the compiler, with complete source code. Compiling PROLOG to run on Forth hardware gives you an inference engine with truly awesome speed. Lou Odette reports that a "naive reverse benchmark" runs at 6,000 LIPS (logical inferences per second) on an NC4016 with a (conservative) 4-KHz clock. When the next generation of Forth processors appears later this year, they should run the same benchmark at the speed of compiled (Quintus) PROLOG on a VAX 11/780.

Flash! At this year's annual Forth Convention (San Jose, Calif., November 1987) Silicon Composers unveiled an IBM PC AT board containing the FORCE core set from Harris Semiconductor. FORCE is implemented on the board as five separate chips: the Forth-based CPU, a hardware multiplier, an interrupt controller, a data stack, and a return stack. All pertinent signals are bused to a prototyping area on the board. It has 32K of high-speed RAM, expandable to 128K (as soon as high-density 35-nsec RAM chips become available).

The development system includes an optimizing compiler that reads standard ASCII text files. You could use this system to prototype and test hardware macros before integrating them into a custom VLSI Forth-based RISC machine. The AT/FORCE board is available from Silicon Composers ([415] 322-8763) starting mid-December 1987 and retails for $4,500.

Text and Data Files

In my last column (December 1987), I looked at a minimal but useful set of string operators, which are summarized in Example 1, page 110. The word *MATCH* replaces the word *–MATCH* in the last column. Its source code is in this month's listing (see Listing One, page 86). –MATCH is the name of a polyFORTH word that compares strings cell by cell and is used primarily for searching index files.

The string package was built on the Standard prelude (see *DDJ*, October 1987) and the *DDJ* Controlled Reference Word Set:

```
2* D2* HEX C,
BL ERASE BLANK .R
2>R 2R> AGAIN DLITERAL
S>D WITHIN TRUE
```

Definitions for these words are also in the listing.

This month's topic is accessing files from Forth. You may have heard that Forth is both a language and an operating system. It's true. Because Forth is often the first development system to work in a new environment, it needs to manage memory, handle interrupts, and access mass storage. The essential primitives for reading and writing mass storage are *BLOCK* and *UPDATE*. Eventually, when other operating systems become available, Forth may be extended to coexist peacefully with them.

When Forth is itself the operating system, it divides mass storage into consecutively numbered (1,024-byte) blocks. Both source code and data are kept in these blocks. There are no "text" or "data" files in the normal sense. Any source or data read by Forth is previously written by Forth.

When Forth runs under an operating system, it takes advantage of operating system calls. Under CP/M, for example, any logical sector in a file can be read or written directly. Sectors can be grouped into blocks, and so any file can be accessed as a sequence of blocks.

But what if the file is not an integer number of blocks long? The file

will only fill part of the last block. This leads to the first rule of standard file access from Forth:

1. *BLOCK* must be able to read the entire file. If the last block is partial, the contents of *BLOCK* past the end of the file are undefined. Reading a partial block is not an error condition.

What about writing to a partial block? The UPDATE command tells Forth only that some part of a block has been written to but not which part. This leads to a second rule:

2. UPDATE forces a partial block to be extended to a full block. The file length must be adjusted accordingly.

If you use Forth only to read Forth files, you will see no partial blocks and neither rule will apply. Otherwise, two operating system calls are required:

1. to determine the length of a file in some unit easily convertible to a double number of bytes
2. to extend the length of a file by a multiple of the same unit

The easiest way to keep track of the length of a file is to call the system when the file is first opened and to maintain its double-number length in a *2VARIABLE* (or equivalent)—for example, *CAPACITY*. Furthermore, assume that you can construct the word *EXTEND*, which extends the file by at least a given double number of bytes. Suppose, for example, that Forth provides the word *MORE*, which extends a file by a given number of blocks:

```
: MORE ( n ) ... ;
: EXTEND ( d )
\ extends file d bytes.
   1024 UM/MOD SWAP
   IF 1+ THEN (round up)
   MORE ;
```

In the interests of keeping primitives primitive, assume that whoever calls *EXTEND* also maintains *CAPACITY*. Some operating systems extend files automatically, so you can use a null definition for *EXTEND*:

```
: EXTEND ( d ) COMPILE 2DROP ;
\ null definition.
   IMMEDIATE
```

BLOCK and *UPDATE* give you direct access to a file. Data stream and text files, however, use sequential access, reading and writing the next group of characters or bytes. This implies that you should maintain a current (double-number) byte position for the file, which can be kept in a *2VARIABLE* (or equivalent)—call it *POSITION*. *POSITION* is initialized to 0 when the file is first opened. The words *GETDATA* and *PUTDATA* can now be defined to read or write the next *n* bytes of a file to or from a memory location:

```
: GETDATA ( a n – n2) ... ;
   \ reads n bytes of data from file
   \ into address, n < 64K. Returns
   \ n2 bytes not read, ie beyond
   \ end of file.
   : PUTDATA ( a n) ... ;
   \ writes n bytes of data to file
   \ from address, n < 64K.
```

High-level definitions of these words are in Listing One. If possible, you should implement them as system calls instead.

Given *POSITION*, *CAPACITY*, *GETDATA*, and *PUTDATA*, you can immediately implement *GETLINE* and *PUTLINE* to access text files. A text file is simply a sequence of text lines, which are in turn a sequence of characters terminated by an end-of-line condition or an end-of-file condition. These conditions are typically implemented as special characters. Each line may be terminated by an optional line-feed character:

```
: GETLINE ( a n – a n2 f) .... ;
\ reads n bytes of text from file
\ into address, n < 64K. n2 bytes
\ are actually read. True if end-
\ of-line terminates line.

: PUTLINE ( a n ) ... ;
\ writes n bytes of data to file
\ from address, n < 64K.
```

GETLINE is based on *GETTEXT*, which is based on *GETDATA*. The end-of-line flag is needed because either end-of-line or end-of file can return a zero-length string.

Note that all the words I've just described can ultimately be built from *BLOCK* and *UPDATE*, provided that the given rules and assumptions are valid. Nothing has been said about file selection, which changes greatly from Forth to Forth. Selecting the proper file before accessing it is up to you. Most operating-system-oriented Forths allow you to select one of at least two files, typically one for input only and one for output or modify. When you change files, you must update or switch *CAPACITY* and *POSITION* accordingly.

Now that you can access text files, you can do some truly wonderful things:

```
: TYPE-FILE
\ reads and prints a file.
   BEGIN PAD 80 GETLINE
   WHILE CR TYPE
   REPEAT 2DROP ;
```

It really is that easy. In the listing, I have included examples for copying text files and for converting blocks to text and back again. One final example shows you how to interpret text files, which you can create using Borland's SideKick or some other handy text editor:

```
: EVAL-FILE
\ reads and interprets a file.
   BEGIN PAD 80 GETLINE
   WHILE EVAL
   REPEAT 2DROP ;
```

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 86.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

EXAMINING ROOM

The Norton Guides

Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later. Hard disk recommended
Pricing:
$50 for the program, $50 for each language guide
Vendor:
Peter Norton Computing Inc. 2210 Wilshire Blvd, Santa Monica 90403 (213) 453-2361

Some years ago, after watching me work, my daughter announced to her playmates that a programmer is someone who sits in front of a computer looking things up in books. It seemed funny at the time, but the more I do it, the more I realize she was right. For the uninitiated, here's how to program: open to the index of manual A, turn to half a dozen references before you find what you need, make two keystrokes, then open manual B and repeat the process. While compiler-makers toil mightily to shave milliseconds off compile time, we plod through weighty tomes checking error codes, parameter-passing conventions, syntax diagrams and all the rest. Simply put, the greatest obstacle to programmer productivity is the everlasting need to look things up.

Well, Peter Norton has done something about it with his Norton's Guides. The glib hype says it's a product for those who hate manual labor. A more specific description is that it's on-line, language-specific help *for serious programmers*. Yes, it's another memory-stealing 37K Terminate-and-Stay-Resident program, but think how much desk space it clears up.

Any time you're in text mode—editing a program, using a debugger, whatever—Shift-F1 wakes up the Norton Guides by giving you a half-screen pop-up containing a menu bar. Thoughtfully, the pop-up appears in the half away from the cursor, so that it doesn't obscure the code you're working on. You can toggle to full-screen with F9, and make the pop-up go away with Esc or F10.

And what do you get with the pop-up? Initially, a list of every instruction in the language on a scrollable panel. You can also do a keyword search that takes you straight to the instruction you want. Positioning the highlight over an instruction, you press Enter and *shazam!* you get a detailed description of its purpose and syntax, often with a code example. There are also cross-references to related statements and topics, to which you can jump by pointing to a pick list entry and pressing Enter.

Additionally, the Guides furnish information about specific features of a given language implementation (the library functions of Turbo C and Microsoft C, for example, as well as data types, reserved words, etc.). Another menu selection called Tables leads to all those little details that consume so much look-up time in manuals: the ASCII values of line-drawing and graphics characters, color codes, etc. In short, the Norton Guides furnish, with a few keystrokes, access to most of the information programmers need while in a fever of creativity.

The foundation of the Guides is a $50 program that makes it work. Then you'll also need one or more of the language databases, each priced at $50. The entry price, then, is $100, plus $50 per language thereafter. If you have more than one database, the Options menu lets you switch among them with a couple

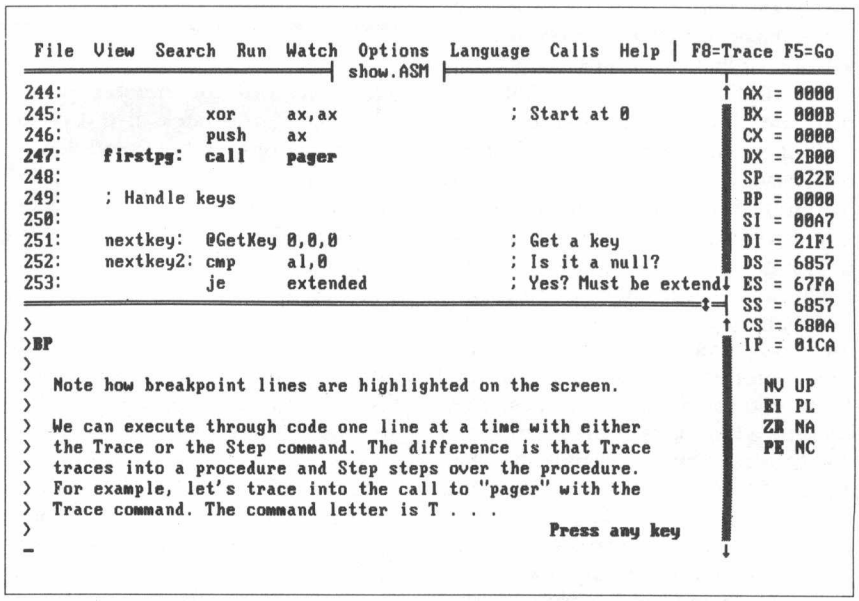


Figure 1: *The Norton Guides add on-line help to a number of compilers, including Turbo C.*

of keystrokes.

As of this writing, Guides are available for BASICA, QuickBASIC, Turbo BASIC, Turbo Pascal, Turbo C, Microsoft C, and MASM. A spokeswoman for the company says more are in the works and should be out by the time you read this: the OS/2 kernel and Microsoft Windows are two that she'd 'fess up to.

For serious programmers, I recommend a set of two databases: MASM and the high-level language you normally use. That's because the Assembler Guide contains a wealth of DOS-related information not found in the others: error codes, interrupts, file attributes, and descriptions of the PSP and FCB structures, among others. If you want to reach deep into the machine from C or Pascal, you need this stuff.

There's a mistake in the Tables, which are common to all incarnations of the Guides. It says the screen attribute byte is given by 256 * background + foreground; replace 256 with 16. I reported it to the vendor, so it should be fixed in future releases. The Turbo C sample program for vprintf() closes a nonexistent stream. Other than that, my only complaint is that the tutorials are cloyingly cutesy. These minor points are enormously outweighed by the convenience of having a comprehensive language reference instantly available at one's fingertips.

In addition to the basic TSR, the program package includes tools for rolling your own guides. Norton calls them a compiler and linker, but don't be misled; they constitute a text formatter governed by a half-dozen commands embedded in the ASCII files. These "bang commands" associate text lines with menus, set up cross-references, establish linkages among text files, and so on. Once you've linked your files, they become a database available through the Guides' Options menu. It takes about 15 minutes to master the whole process.

This opens the way to tremendous possibilities for application developers. Instead of laboriously constructing help screens and all the code required to manage them, you can write a Guides database and bundle it with your software. Norton Computing says they're willing to license distribution rights to the underlying TSR program.

The Norton Guides don't completely eliminate the need for manuals and reference books, but they reduce clutter around the keyboard. They also let you write code a whole bunch faster, which is good news. This product represents a quantum jump in programmer productivity.

by Kent Porter

Soft-Scope Source-Level Debugger

Target:
PC/XT and compatibles
Required:
DOS 2.0 or later, hard disk, 256K. Compiler/linker generating Intel-type symbols
Price:
(DOS version only, others available) Debugger is $500 One-year update/support is $100
Vendor:
Concurrent Sciences Inc. P.O. Box 9666 Moscow, ID 83843; (208) 882-0445

Throw out the quiche, put the dog in the back of the pickup, and crack open a beer; Soft-Scope is a real man's debugger. No wimpy windows, no frilly function keys, no cuddly colors. No Sir, this is purely a command-driven program on a monochrome screen that keeps on scrollin'. It might be argued that real men don't use symbolic debuggers at all, but those who do will appreciate the full features and solid feel that underlie Soft-Scope's rugged exterior.

Soft-Scope reminds us of the forgotten fact that software doesn't have to be cute to be useful. What it lacks in sex appeal, this debugger makes up for in functionality. It monitors and reports program execution at the source level in any of Pascal, C, FORTRAN, Assembler, and even PLM and JOVIAL. You can set breakpoints on source lines and refer to variables by name when checking or assigning values. If you want to see what the compiler generated from a source line, you can view the resulting machine language as assembly code.

The point of reference when working with Soft-Scope is the source line number, which it gets from the requisite .LST file. Like a BASIC interpreter, it tracks its position in the source file via line numbers, and lists ranges of lines on command. It also associates a line number with a subroutine name, so that you can refer either by line or by name to entry points for setting breaks and the like. A command lets you find out the range of lines for a given routine. This is useful for setting a watchpoint within that range only, using the commands "RANGE 100 120" and "GO TIL X = 5."

The single-step trace mode works by default at the source line level. You get into single-step with the S command, then keep punching the space bar to execute successive lines. Soft-Scope displays each line as it executes it. If you get tired of this, you can hit Enter instead of the space bar, then specify a speed from 0 (slowest) to 9 (fastest), and the debugger runs in continuous mode, displaying each line as it runs. The Q key (or alternatively Esc) stops it so that you can examine variables. You can also disassemble the source line and single-step at machine-code level, which is a nice feature for uncovering those bizarre glitches that sometimes crop up in programs that tickle the deep innards of the machine.

Soft-Scope has the particularly attractive ability to display the nesting of calls by name. A program crashes in some subroutine and you wonder how on earth it got there; this tells you who called whom, back up to the main program level. You can also list the stack itself to any level from the top down; e.g. "20 STACK"

(continued from page 121)

lists the top 20 words. Another command lists the stack size and the high-water mark, handy for diagnosing crashes due to stack overrun.

This debugger understands aggregate data structures. You can display a structure by name, listing its fields and data types, and fetch any of the fields' values. The same is true of arrays, which can be dumped as a whole, by row, or by individual element.

One of the best features of Soft-Scope is pointer dereferencing. This allows you to follow an indirection chain by dereferencing a series of pointers to an object, whose value appears on the screen. You can also view the resolved address to find out if you've got a screwy pointer. The pointer dereferencing notation is pure Pascal, which might confound those without a Pascal background. Still, it's not difficult to master. If INTPTR is a pointer to an integer, then the value of the integer is represented by INTPTR^. Similarly, STRUCT^.FLD^.OBJ is an indirection chain in which STRUCT points to FLD, which points to OBJ.

Another feature worth mentioning, is Soft-Scope's ability to route all of its prompts and displays to an external RS-232 terminal. This feature makes the program well suited for debugging programs which are highly interactive with the screen cursor position or are graphical in nature.

Soft-Scope's primitive user interface is usually not a problem. Indeed, elaborate displays such as CodeView's often dazzle the eye and thus get in the way. It would be nice, though, if you didn't have to halt the trace in Soft-Scope and type a command to examine registers and variables. Also, the inability to switch display pages in Soft-Scope means that program and debugger output get mixed together, which can lead to a wild jumble.

Soft-Scope has 37 commands, most of which have several optional modifiers. The program acknowledges human frailty to the extent of furnishing a HELP command that accepts an argument. For example, "HELP SUBMIT" furnishes a synopsis of the command syntax for exercising a debugger script file.

But until you're a certified Soft-Scope guru, you can't effectively work this debugger without the manual at your elbow. The documentation is readable if a bit terse. There are 305 pages of tutorials using several programming languages, and the command reference is 98 pages. A major gripe with the manual is that it lacks an index.

Soft-Scope comes in several flavors. The one I reviewed is for MS-DOS. There are also incarnations for the likes of RTCS/UDI and iRMX. Prices range from $500 for the DOS version up to $2000 for the Intel 310 with iRMX 286 OS, and considerably higher than that for versions working with the Applied Microsystems ES 1800 emulator. Clearly, this is a debugger for serious developers.

Despite its no-frills, macho interface, Soft-Scope delivers for those who can afford it.

by Kent Porter

XO-Shell

Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later
Pricing:
$49
Vendor:
Wyte Corp.
701 Concord Ave,
Cambridge, MA 02138; (617) 868-7704

Every once in a while you run across a smoothly crafted little program and you just hate the guys who dreamed it up: Why couldn't I have thought of that? The only consolation is that it's probably almost as much fun to use as it was to write it.

XO-Shell is such a program. I'm not sure what to call it. It's not really a debugger, nor is it exactly a DOS shell. The vendor refers to it as a programmer's productivity aid. Okay, but it could be a productivity tool for non-programmers, too. Agreement on one point is assured: XO-Shell is slick. It does a lot of stuff that you wish DOS and/or your editor would do.

I tend to have an allergic reaction to TSR's (Terminate-and-Stay-Resident programs). They can so overload memory with conveniences that you don't have enough left to do real work. And so I viewed XO-Shell with a certain wariness, especially in view of its 88K requirement. To exact that kind of price, I figured, it'd better be good.

Your jaundiced reviewer was pleasantly surprised. There's a whole host of goodies that make that 88K worth the cost. On the more prosaic side, XO-Shell stores up DOS commands for recall, edit, and re-execution, and it has a file viewer similar to Xtree and other DOS shells. What sets XO-Shell apart from other shells is that you can do this stuff from within an application such as an editor or the Turbo/Quick environments. But that's not all.

Say you're editing a program module and you've forgotten the exact spelling of a variable defined in a different module. With XO-Shell, you can pop up the other module and check it, or print out a portion of that code, without leaving your edit session. Whoops! You just noticed an error in the other module. No sweat, XO-Shell lets you correct it on the spot, then escape back to your main editing job. All it takes is the Alt-n hot key and a few keystrokes.

Last month you wrote a nifty little algorithm, and now you need to plug it into your latest masterpiece. Position the cursor where you want it to go, bring up that old file with XO-Shell, and mark the text with some Wordstar-like commands. When you leave XO-Shell, it copies the marked text into the new program, effecting a clipboard.

Maybe you remember the name of that algorithm, but not the name of the source file containing it. XO-Shell has a grep-like facility to search file groups for a specific string. Function keys direct the search results to the screen, the printer, or a file,

and you can save the files list for future use in wide-ranging searches. XO-Shell also searches the file currently being worked on, in case your editor lacks this elementary capability.

Few things are as annoying as having to leave an edit session and list the directory simply because you forgot the name of an include file or data file. XO-Shell lets you list a directory while editing. It can also search directories to find a specific file if you can't remember which sub it's in. And if, while browsing among files, you decide it's time to clean up some directories, XO-Shell has the ability to copy and erase. When you're done, hit Esc and resume editing.

So now the program is done, but there's a variable name you want to change, or maybe one that's declared but possibly not used. Pop into XO-Shell and do a cross-reference listing among the program modules. It lists every line, identified by file, where the variable occurs.

There are other things as well. For example, XO-Shell can save output screens from your program in a disk file, and restore them later for double-checking. For another, there's a pop-up via Alt-m that shows all 256 characters; pick one from the display and plug it into your edit screen. Even if your editor can't generate graphics characters, you can incorporate them into the text this way. And XO-Shell lets you drag them around and replicate them, providing for text boxes and other visual effects in source listings, as well as direct encoding of special characters in string literals.

Aside from the hoggish 88K, my only criticism of XO-Shell is that the helpful hints on the prompt line (which appears only in response to the hot key) aren't very helpful. Such things as "F9:Chng_para" are far from intuitive, even for C cowboys.

XO-Shell has won its way into my programming toolkit, and I recommend it.

by Kent Porter

(Examining Room continued on page 130.)

Microsoft Macro Assembler 5.0

Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later, hard disk recommended. Minimum 256K memory, 320K recommended
Price:
$150
No licensing restrictions on generated code
Vendor:
Microsoft 16011 NE 36th Wy, Box 97017, Redmond, WA 98073 (206) 882-8080

The latest release of Microsoft's venerable MASM brings a new level of productivity to this most tedious of all programming languages. It's loaded with new features, most importantly a bundled copy of the CodeView symbolic debugger.

Microsoft claims that MASM 5.0 assembles 25 percent to 40 percent faster than 4.0. I didn't validate this with any test suites, but it is screamingly fast on an AT: under two seconds for a 150-line program and about five for an 800-liner, which is noticeably faster than earlier versions of the assembler.

With release 5.0, MASM now supports the instruction sets for all PC processors up through the 80386/387. The default mode of the assembler is the 8086 instruction set; pseudo-ops such as .386 and .387 tell the assembler to honor instructions for the indicated targets. If you forget to include the appropriate directive, the assembler scores a severe error for each non-8086 instruction and issues a complaint.

Speaking of pseudo-ops, one potential gotcha for those migrating upward to 5.0 has to do with the encoding of floating point numbers. Before, MASM encoded real numbers in Binary format. The default in 5.0 is the IEEE floating point standard, compatible with 80x87 math

coprocessors. If you still want Microsoft binary format, you have to assert the new .MSFLOAT directive.

Yet another pseudo-op called .MODEL implements memory models, new with MASM 5.0. Its operands are SMALL, MEDIUM, COMPACT, LARGE, and HUGE, which correspond to the models available in high-level Microsoft languages such as C and Pascal, as well as in Turbo C and, to a lesser extent, Turbo Pascal 4.0. This addition simplifies the assembler interface to other languages, since it generates pointers of the correct size for the high-level model being used. Clearly, the emphasis is shifting away from assembler as a primary language to assembler as a language for linked high-performance subroutines, and this addition to MASM recognizes the trend.

To that end, MASM 5.0 furnishes a macro library for mixed-language programming. This file, MIXED.INC, includes macros for passing and receiving arguments, allocating local variables on the stack, segment fixups, exit processing, and other high-level interfacing needs. The macro library is propped up by an excellent 140-page guide covering a range of mixed-language programming issues.

The overall quality of the manuals is much better than any previous Microsoft documentation. In addi-

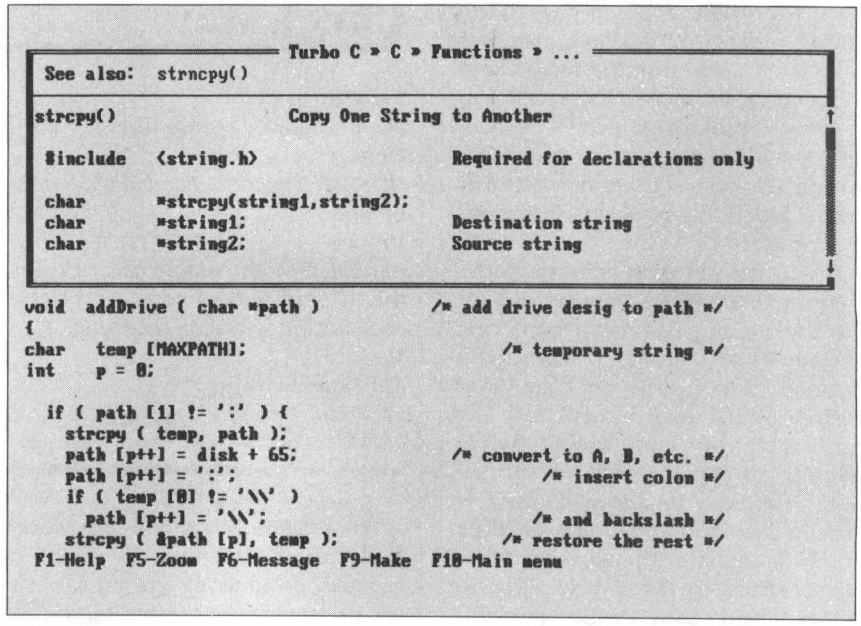


Figure 2: *MASM 5.0 includes Microsoft's CodeView debugger.*

tion to the mixed-language guide, there are two soft-bound books, a 467-page Programmer's Guide and a 401-page volume covering CodeView and other utilities. Rounding out the documentation is a 148-page wire-bound reference divided into five tabbed sections. Intended to be kept at the programmer's elbow, this reference summarizes every instruction, pseudo-op, and command-line switch. There are several brochures and a function-key template for operating CodeView.

If you haven't worked with CodeView before, you're missing out on the finest thing Microsoft has done to date. This is how God intended a symbolic debugger to be. It's a wonderfully visual environment that lets you watch your code execute while keeping an eye on registers and selected memory locations. CodeView runs in a secondary video page, thus not interfering with programs that do graphics or fancy text screens, and you can easily toggle back and forth between the CodeView display and your program's output. Accommodating to all and sundry, CodeView lets you control it via pull-down menus, function keys, typed commands, or a mouse: your choice.

An example of the possibilities with this debugger is a thing called a watchpoint. You can tell CodeView to watch a memory location or even a register and, when it changes to a specified value (or changes at all), CodeView halts and reports the condition. If you've got several watchpoints and breakpoints set, the CodeView display twinkles like Times Square on Friday night, but without the chaos; you are actually watching your program run.

CodeView is not specific to MASM 5.0. It works with all of Microsoft's flagship languages: C, FORTRAN, Pascal, and QuickBASIC. All that's required is to compile and link with certain options, and then you can debug an entire application written in any combination of these languages. Alas, it doesn't work with Turbo languages, but that's hardly surprising.

MASM 5.0 doesn't reduce the notoriously finicky detail of programming in assembly language, but it makes for more productive tedium.

by Kent Porter

CheckMate

Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later, hard disk recommended
Price:
Annual fees for single-user, department, and site licenses start at $5,750; demo version available.
Vendor:
Cinnabar Software
2704 Rio Grande, Ste. 1, Austin, TX 78705 (512) 477-3212

CheckMate is a quality assurance tool which runs on a PC or compatible allowing a tester to easily set up automatic scripts for testing communications. CheckMate is intended to test the host computer or communications service (hardware or software) to which the PC is attached.

At the highest level, CheckMate consists of a terminal program. Unfortunately, CheckMate does not emulate any of the popular intelligent terminals such as the VT100 (this feature is scheduled for inclusion in the summer of '88). It does, however, provide a lot more control than is normal with such programs. All communication parameters are accessible. In addition, a high level screen log and a detailed screen trace record all that is happening, both from the keyboard and from the communications ports. Turn your modem off or drop carrier and its instantly recorded on the trace.

CheckMate also provides a capture mode. In this mode, the user goes through a normal communications sequence. CheckMate makes detailed notes on what was entered at the keyboard and what was received at the COM ports, writing this information into a "script" file. During replay, CheckMate makes the keyboard entries and compares the results with what it expects. CheckMate also notes changes in line status such as Carrier Detect or changes in baud rate. The executing sequence can be viewed in progress from the trace screen. Deviations from the expected are noted in the log file. Both the log and the trace can be recorded on disk for later perusal or as an testing audit trail.

CheckMate is designed to allow the user to set up scripts for testing every feature of his product. With such scripts, the user can build a test suite for acceptance testing of the program or service. This suite can be built upon as each new generation is developed. In this way, the tester is assured that no existing features have been "broken" by new additions. These scripts can also be used as a routine status check or to "attack" the system, running the same sequence over and over until a failure occurs.

Unlike most other communications software, however, CheckMate's script files are recorded in Microsoft 4.0 compatible C source code (an example script, created by CheckMate but with comments provided by Cinnabar, appears in Listing One). To play a script back, the user must exit CheckMate, compile and link the C program into an executable file that can then be played back from within the program. CheckMate is intended to handle large, carefully designed scripts which the user might run hundreds of times. The time to compile might be a nuisance during the actual development if it takes multiple iterations to get the script right.

For this, the user gets some pretty significant advantages. Compiled scripts run faster with better timing control than an interpretive script can. Roughly half of the user manual is devoted to a description of the system calls available to the CheckMate programmer. In addition, CheckMate's scripts can be edited by a C programmer to add any extra features desired. The complete C language is at the programmer's disposal.

While not for everyone, CheckMate should be an interesting prod-

uct to those companies which provide communications services or which make heavy internal use of serial communications. CheckMate can also be used to test mainframe software by simulating multiple users logged into serial terminals. CheckMate comes on two floppies within a 8.5-by 11-inch 3-ring binder. The sitngle CPU lvicense fee for CheckMate is $5,750 per year. A demo version can be ordered for $49.95. CheckMate requires 320k and DOS 3.1 or better to run.

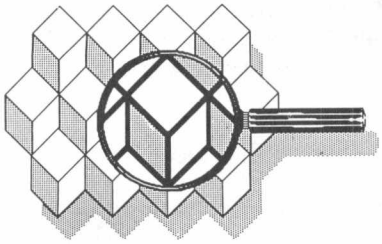
by Randy Davis

DDJ

```
Check*Mate header      */

main(argc, argv)
int argc;

char *argv[];

{
  int  length;                        /* Declare program vaiables.      */
  int  prevwait;
  /* Set online requirements for the Hayes modem.                        */

    PM_set(COM1, _SXDCE, SXCTS);
 /* Set error action parameter to exit the script if any errors occur. */

 PM_set(COM1, _ERRACTION, EAEXIT);
 SX_assert(COM1);                    /* Raise dataset signals and...       */
 MX_online(COM1);                    /* match for online on COM1.          */
 TX_str(COM1, "AT", YES);            /* Send Hayes attention command...    */
 MX_blk(COM1, "OK", 2, 10, &length);/* and match for "OK" returned.        */
 CM_delay(0.5);                      /* Wait for Hayes modem to catch up. */
 TX_str(COM1, "AT DT", NO);          /* Send Hayes command to dial tone    */


 /* Send the phone number for the modem to dial, which is accessed by     */
 /* the script with the string 'argv[1]'.                                 */


 TX_str(COM1, argv[1], YES);

 /* Match "CONNECT" back from modem signaling that host answered.         */

     PM_read(COM1, _WAIT, &prevwait);

     PM_set(COM1, _WAIT, 60);

     MX_blk(COM1, "CONNECT", 7, 40, &length);

     PM_set(COM1, _WAIT, prevwait);

     exit(OK);                              /* Exit script with success      */

}
```

Example 1: *C source code generated by CheckMate.*

OF INTEREST



Debuggers

Gimpel Software has announced Turbo C-terp, a debugging C interpreter that is fully compatible with Turbo C. Breakpoints can be temporary or sticky; can be specified via line number, function name, or cursor; and may be conditional. Users can step to the next statement, step over functions, and leave the current function; display the value of any expression; and execute any C expression. C-terp supports full K&R (with ANSI enhancements) and multiple modules and includes a built-in multifile editor, automatic make, virtual memory option, and shared symbols. The product is priced at $139. Reader Service No. 16.
Gimpel Software
3207 Hogarth Ln.
Collegeville, PA 19426
(215) 584-4261

dBFind is a dBASE II, dBASE III, and dBASE III Plus debugger from **The Software Development Factory**. The debugger identifies missing keywords, arguments, and operators; locates unbalanced control structures; builds a complete index of variables by source module and by line number and includes type of usage for each occurrence; and provides command-specific, detailed error messages. The price for dBFind is $99. Reader Service No. 17.
The Software Development Factory
400 E. Pratt St., Ste. 800
Baltimore, MD 21202
(301) 666-8129

Arium Corp. has introduced a high-level C language symbolic debugging feature for its Echo microprocessor development system C compiler option. This feature allows design engineers or programmers to debug emulated code in the source language, in assembly language, or both. The C compiler option with C language debug feature is available for $495 for the 8-bit unit and $895 for the 16-bit unit.

Echo is a microprocessor development system that features a 20-Mbyte hard disk, a 1.2-Mbyte flexible disk, 1 Mbyte of main memory, a full-screen editor with dedicated keypad, assemblers, linker and loader software, and complete emulation software. Echo is priced at $8,960 for 8-bit processors and at $12,980 for 16-bit processors. Reader Service No. 18.
Arium Corp.
1931 Wright Circ.
Anaheim, CA 92806
(714) 978-9531

Tools and Utilities

BPTPLUS in C is a B+Tree data manager available from **Sterling Castle** that allows both direct and sequential access to data. The program features multiple nonunique keys, virtual-index capabilities, partial key searches, and the ability to include 40 indices per index file. With the program, users can convert BASIC programs written with B+Tree to C programs and have access to original BASIC index and data files without having to reenter data. BPTPLUS is compatible with Borland's Turbo C; Microsoft C, Versions 3.0 and 4.0; and Lattice C, Version 3.0. Reader Service No. 19.
Sterling Castle
702 Washington St., Ste. 174
Marina del Rey, CA 90292
(800) 722-7853
In Calif. (800) 323-6406
(213) 306-3020

The Cito 286 programming environment is now available from **Fillmore Systems**. Cito consists of an interactive command-line interpreter and macro language and a linking feature that allows object modules to be dynamically linked into the currently running Cito environment. Users can write, edit, compile, and link C procedures from within the Cito environment. The macro interpreter generates executable machine code for macro definitions and entry points to dynamically linked procedures. Cito supports conditional testing, high-level control structures, file I/O, multitasking, variables, and arithmetic. Version 1.0 of Cito runs on IBM PC ATs and compatibles using Xenix, Version 2.0 (System V). The price is $229. Reader Service No. 20.
Fillmore Systems
7200 York Ave. S., Ste. 301
Edina, MN 55435
(612) 831-6984

The C-scape On-Line Reference has been jointly released by **Oakland Group** and **Peter Norton Computing**. The software is designed to work in conjunction with the Instant Access Program of the recently released Norton Guides and Oakland's C-scape Interface Management System. The Norton Guides provides instant on-line reference data for four languages: C, assembly language, BASIC, and Pascal. C-scape is a professional development tool for controlling the user interface of C programs. The C-scape On-Line Reference Guide is available from Oakland Group for $50. The program requires the Norton Guides' Instant Access Program, which can be purchased from Oakland Group or Peter Norton Computing for $50. Reader Service No. 21.
Oakland Group Inc.
675 Massachusetts Ave.
Cambridge, MA 02139-3309
(800) 233-3733
(617) 471-7311

C:Lines/C:Tree, from **SoftRex**, is a program that reorganizes and prints C source code and provides a structural analysis of the code. Features included are outlines of all logical blocks that are added to the listings, detailed cross-references, an automatic source code reformatter, and a hierarchy tree that shows the relationships between the various subroutines in code. C:Lines/C:Tree can

also generate a listing of unused global symbols, preview listings on a monitor, analyze interactions between overlays, add titles and subtitles to listings, generate an index of titles and subtitles, and be configured for any printer. The program runs on IBM PCs and compatibles and sells for $59.95. Reader Service No. 22.
SoftRex
4807 Bethseda Ave., #287
Bethseda, MD 20814
(301) 881-8274

A new coding tool from **O'Neill Software** called the Text Collector lets programmers find, collect, examine, and analyze scattered blocks of source code. The package searches an entire disk or specified groups of files for any combination of terms and automatically saves all context blocks meeting the search criteria. Context blocks can be lines of code, strings, comments, records, or other user-defined blocks of material. File names can be appended or prepended to blocks, and the blocks can be sorted. The program permits complex searches using Boolean operators, nested parentheses, and 14 different wildcards. The Text Collector runs on IBM PCs and compatibles with 128K and MS-DOS/PC-DOS 2.0 or later. The price is $69. Reader Service No. 23.
O'Neill Software
P.O. Box 26111
San Francisco, CA 94126
(415) 398-2255

The Window Management System (WMS) for Turbo C is a high-level programming tool that allows efficient development of easy-to-use interfaces based on windows, menus, and data prompts. **B&C Microsystems** is now shipping this program, which features a window editor, a window librarian, a C interface, and DOS utilities. WMS sells for $69.95. A demo disk is available for $5 and can be credited toward future purchases of WMS packages. Reader Service No. 24.
B&C Microsystems
355 W. Olive Ave.
Sunnyvale, CA 94086
(408) 730-5511

Serengeti Software has announced Show Me!, a memory-resident, file-viewing utility for the IBM PS/2, IBM PC, and compatibles. The program allows users to list directories and view up to four files instantly; use pop-up help screens for virtually any program; look at WordStar document files while editing and pass text from a window; view multiple source or listing files in ASCII or hexadecimal while using DEBUG, SYMDEB, or other utilities; and add help screens to programs using built-in program interfaces to BASIC, Pascal, dBASE, and C. Show Me! sells for $39. Reader Service No. 25.
Serengeti Software
P.O. Box 27254
Austin, TX 78755
(512) 345-2211

Languages

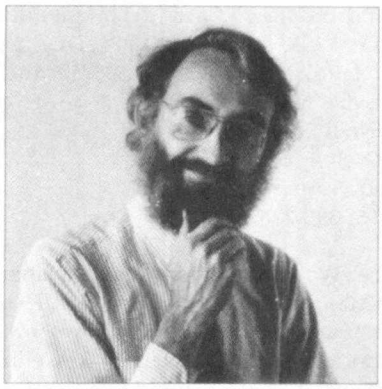
Addison-Wesley has released *The AWK Programming Language*, the first book on awk, a programmable text-manipulation language for writing short programs to perform data-manipulation tasks. The language was originally developed by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger in 1977. A new version of the language, which is available for Unix and MS-DOS, was developed in 1985. Some of the topics covered in the book include retrieving, transforming, reducing, and validating data; managing small personal databases; text processing; little languages; and experimenting with algorithms. The book sells for $21.95. Reader Service No. 26.
Addison-Wesley
Reading, MA 01867
(617) 944-3700

MHT Software is now shipping three new products: the cENGLISH Menu Development System; cENGLISH, Version 3.0; and cENGLISH for Microsoft C. The cENGLISH developer employs English-like commands that are translated into C language source code. This C code uses the industry-standard C-ISAM access manager and in Informix-SQL database management system to organize and retrieve information. The cENGLISH Menu Development System is an interactive utility for building many different styles of menus. cENGLISH, Version 3.0, adds video and printer control, text file manipulation, and utilities for importing and exporting ASCII data. Programmers without C experience can use cENGLISH for Microsoft C and benefit from the advantages of the C language. Prices vary according to machine and operating system. Reader Service No. 27.
MHT Software
2923 Saturn St., Ste. A
Brea, CA 92621
(714) 528-1602

Software Development Systems has announced its UniWare Z80 C compiler, which was designed to help engineers make the transition from writing in assembly language to writing in C. A special optimizing feature of the UniWare Z80 C compiler is its ability to use the Z80's alternate register set for storing up to three 16-bit user-declared variables. The compiler comes complete with a macro relocating assembler, a linker, an object module librarian, and utilities to help programmers debug their code and download it into emulators and EPROM programmers. The UniWare Z80 C compiler running on the IBM PC and compatibles using MS-DOS or Xenix is priced at $995. The compiler running under Unix on a variety of machines is priced at $2,995. Reader Service No. 28.
Software Development Systems Inc.
3110 Woodcreek Dr.
Downers Grove, IL 60515
(312) 971-8170

DDJ

SWAINE'S FLAMES



True magazines really do have something like personality. You can develop a relationship with them. That's why it seems so unfair when a publisher repositions an old friend, as McGraw-Hill is doing with *Byte*. It's all the more annoying when the change seems so patently unnecessary. Isn't it obvious that we need a Scientific American of computer technology more than we need another PC magazine? Not to McGraw-Hill it isn't.

McGraw-Hill is not alone in opting for the product-catalog approach; the trend is depressingly widespread. Even our new section of short reviews, Examining Room (debuting in this issue), could be seen as symptomatic of the trend. But *DDJ* is, I assure you, in no danger of becoming a product catalog. It wouldn't fit with the personality.

Speaking of personality, the Peter Norton T-shirt came in the mail recently, close on the heels of the Peter Norton poster, the Peter Norton Christmas card, and the Peter Norton coffee cup. (If you didn't like the coffee cup, Peter Norton would send you a Peter Norton hammer to smash it with.)

Now I believe that the Norton Utilities are nifty, and that Peter has as much right as anybody to throw his hat in the ring for Mr. Cultural Icon of 1988, but all this fuss over a company that produces programs for wiping files and cleaning up directories gives me pause. Norton is, after all, in the utility business, a sort of electronic Tidy Bowl man, the digital descendant of Ed Norton. Is this an image you want to display on your coffee table? Consider well when marking your ballot.

Another clean-cut kid, Rob Dickerson, has been making news lately, in the sections of what newspaper business pages call Transitions. Some of you know Dickerson from Microsoft's language division. In December he became Borland's latest acquisition, with the new title of vice president, product marketing. Microsoft responded to Dickerson's career move with a lawsuit and a restraining order, and Borland countersued. By the time you read this the suits should be settled, or at least moot.

Microsoft had reason to be upset; Dickerson knows a lot that could be useful to the competition, and that means Borland. (I am specifically not referring to his knowledge of Microsoft strategies. I have no knowledge regarding Rob's knowledge about Microsoft strategies, thank you, Microsoft lawyers. What I have is informed opinion.) Meanwhile, Borland is building new quarters in Scotts Valley to accommodate its growth. I think it is going to call the new space the Borland campus.

I spent many pages in this magazine last year campaigning for articles on how to break various bandwidth bottlenecks. Bandwidth is, unfortunately, becoming a vogue word. You can find it in John Sculley's *Odyssey: From Pepsi to Apple*, where Sculley applies it to marketing. He probably picked up this usage from Steve Jobs, a good man with a metaphor.

You can find the term used more precisely and with more purpose in *The Media Lab* by Stewart Brand, a good book on current research at MIT into the convergent technologies of publishing, computing, and broadcasting. The principals in *The Media Lab* believe that bandwidth reduction is the issue of the decade, and they are acting on that belief. I recommend Brand's book. (I haven't been inspired to read Sculley's yet.)

I sometimes think that Marlin Ouverson, a former *DDJ* editor, timed his leaving so that he could title his farewell editorial, in paraphrase of Richard Henry Dana, "Two Years Before the Masthead."

Which leads me to the following.

I became editor-in-chief of *Dr. Dobb's Journal* in 1984 and have now spent more years at the helm than any other editor (including Marlin Ouverson). While I am fond of this magazine and think it serves a need that the product-catalog magazines don't, four years is a *long* time—a third of the life of the magazine, a tenth of my own life, longer than Sculley has been at Apple (or about as long as there has been a Macintosh product line)—and these four years have been four years away from full-time writing on the rock pile of a largely administrative job.

For a writer, that's a long sentence.

So, over the next few months, I will be creating a new role for myself with the magazine, one that will allow me to do more writing. I plan to research and write about topics close to my interests and, according to our reader survey and to my informal survey in the November issue, close to yours as well.

More about that next month.

Michael Swaine
editor-in-chief

#137 MARCH 1988

2.95 (3.95 CANADA)

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

OBJECT-ORIENTED DESIGN

TOOLS FOR HANDLING

- Binary Trees
- Huge Arrays
- EGA Fonts

REVIEWS

CodeView

Turbo Pascal 4.0

03

0 38351 16562 8

ARTICLES

Object-Oriented ▶ **Object-Oriented Programming and Database Design** 18
by Jacob Stein
Object-oriented programming has been gaining acceptance in the last few years. In this article, Jacob gives an introduction to object-oriented programming in general, a description of a specific object-oriented language (OPAL), and some example problems that are particularly well suited to object-oriented solutions.

EGA ▶ **Writing Custom Display Fonts** 36
by Andrew J. Chalk
Andrew describes a TSR he's written for the EGA that automatically replaces the standard screen font with a resident custom font.

C routines ▶ **Threaded Binary Trees** 42
by James Mathews
Textbook discussions of threaded binary trees are almost always accompanied by source code in Pascal, aimed at the university student. In this article, Jim describes some routines he developed for traversing and managing these trees in Microsoft C.

REVIEWS

EXAMINING ROOM 116
coordinated by Ron Copeland
Products examined from the programmer's perspective. This month's offering includes a detailed review of Borland's Turbo Pascal 4.0, and a quick look at Microsoft's CodeView debugger.

COLUMNS

TO THE MACS 90
by Stan Krute
Stan winds up the CDEF project he began in the January column, and talks about several products.

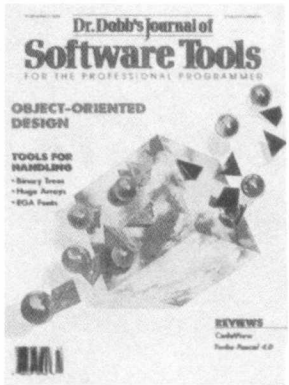
Pascal ▶ **STRUCTURED PROGRAMMING** 108
by Kent Porter
Not all problems are polite enough, or trivial enough, to shoehorn into the PC's 64K segments. This month Kent shares his techniques for handling large data arrays in Turbo Pascal.

FORUM

EDITORIAL 6
by Tyler Sperry
RUNNING LIGHT 8
by Tyler Sperry
LETTERS 12
by you
CARTOON 14
by Joe Sikoryak
SWAINE'S FLAMES 136
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX: 81
Where to go for more information on products.
OF INTEREST 128
Product news for the programming community.



About the Cover

If nothing else, object-oriented languages have gotten a rap for being abstract and not as efficient as needed for real-world problems. This month's lead article discusses object-oriented programming languages and some real-world problems where the object-oriented approach excels. Sometimes all you need are the right tools (like data hiding and inheritance) and a little "blue sky" thinking.

Next Issue

The annual AI issue, with language hacking techniques for fanciers of LISP and PROLOG, a review of Objective-C, and all the usual foolishness.

EDITORIAL

Object Lessons

Recently I attended Microsoft's annual Systems Software Seminar. This is less a seminar in the academic sense than a carefully orchestrated presentation for the press, a gathering where Microsoftians like Bill Gates and Steve Ballmer take turns at a podium and argue over exactly when OS/2 will take over the known universe. There were also presentations on future enhancements to DOS, the OS/2 LAN and SQL servers, future developer goodies, etc.

Bill Gates remarks on programming languages really got me thinking, though. He spent a fair bit of time explaining to an audience of industry analysts and press people about these things called "objects" and how they were the coming thing in programming. This is a view I quite agreed with. The increasing complexity of environments like the Mac Toolbox and the OS/2 Presentation Manager cry out for new programming tools and methods. Language packages like Actor and Smalltalk/V have the potential (in their inevitable protected mode versions) to be the next wave in programming.

But Bill Gates wasn't talking about Smalltalk or Actor.

The people at Microsoft, you see, have a language division that's expected to turn a profit. It isn't their job to convert the world to new languages, it's to sell language packages. Preferably more than Borland does.

So when asked how Microsoft was going to address the topic of object-oriented programming with its languages, Bill Gates wasn't interested in weirdo languages like Smalltalk; he talked about adding object-oriented extensions to existing Microsoft languages: C and BASIC.

Object-oriented BASIC! What a concept! Soon beginners will be able to add modular meatballs to their spaghetti code. My mind boggled with the possibilities and almost missed the follow-up from Gates: there's no promise of when these extensions will appear, or in what form. ANSI C is still the chosen language, and converting to C++ seems unlikely.

The next day on the plane home I was still pondering the notion of what—for want of a better name—I'm calling OOBASIC (pronounced "uh-oh BASIC"). The question I was wrestling with is non-trivial: how much of the increased sales of programming languages is based on advances in programming technology, and how much is simply improved marketing? It's a difficult question. After all, Borland had a Modula-2 compiler years ago and dumped it. Yet the same company has managed to make money on PROLOG by hyping it as "the language of artificial intelligence."

I've spent some time considering Bill Gates' opinion that no new languages will make an impact in the years to come, and I'm still not convinced. It seems to me that the new paradigms and new environments will demand new approaches, not patches on old languages. The folks at the Whitewater Group and Digitalk and other companies seem to be able to do quite well marketing solutions rather than trying to jump on the bandwagon and sell yet another ANSI C compiler. Just maybe they're on to something. And maybe object-oriented extensions to older languages are really a step back instead of forward. The next year or so will tell.

In the meantime, I'll be spending my evenings exploring Smalltalk and other languages, examining new paradigms, and counting myself lucky on the days I don't get a press release announcing object-oriented COBOL.

Tyler Sperry

Tyler Sperry
editor

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editor *Rhoda Simmons*
Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Artist *Bunny Carter*
Circulation
Circulation Director *Maureen Kaminski*
Fulfillment Coordinator *Francesca Martin*
Subscription Supervisor *Kathleen Shay*
Newsstand Coordinator *Sarah Frisbie*
Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 81*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

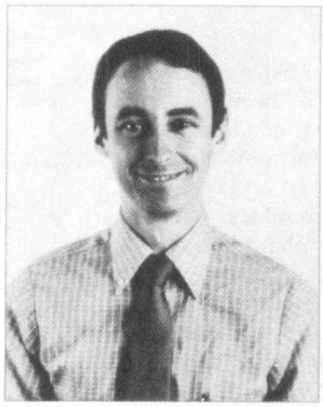
Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT



First things first. Please note that the lack of a "C Chest" column this month doesn't mean we don't love C anymore. It's just that Allen Holub got involved with a few too many projects and had to take a vacation to decompress. Allen will be back next month, fully recovered. In the interim, this issue has plenty of C code to keep you busy.

Next topic: programming books. The floodgates have just started to crank open, and we're about to be inundated with books on programming for OS/2. Many of those books will be nothing more than boring regurgitations of the existing Microsoft documentation. Happily, I have seen some outstanding books lately, books written by programmers who really know what's going on inside OS/2, and I'd like to share the good news.

First up is Ed Iacobucci's *OS/2 Programmer's Guide* (Osborne McGraw-Hill, 1988). Iacobucci was the leader of the IBM OS/2 design team and clearly knows his stuff. This book is one of those (like Peter Norton's excellent *Programmer's Guide to the IBM PC*) that attempts to give you both the massive amounts of information that you really need and yet still be readable. At 1,100 pages, the book certainly is complete in coverage, and Iacobucci covers the material with a simple, clear style. This book is obviously the work of someone who's spent a lot of time helping other programmers get their programs running under OS/2, and wants to share his experience. Unlike most paperback computer books these days, this one's a steal at $25.

Mere recommendations aren't really good enough for *Inside OS/2* by Gordon Letwin (Microsoft Press, 1988). Let me put it more directly: if you're at all serious about OS/2, you *must* buy this book. Letwin was the chief architect for Microsoft's OS/2 team, and with this book he's managed to pull off the nearly impossible trick of explaining the inner workings of OS/2 along with the reasoning behind the architecture, while actually making the book fun to read. If you're new to OS/2, reading this book will take weeks off your learning curve. It isn't often that we're treated to a book this good.

Finally, if you're still writing code for MS-DOS instead gearing up for OS/2, I'm sorry to report that you're probably going to spend a great deal of money soon. The *MS-DOS Encyclopedia* is finally out, and it's generally a splendid (and massive) volume. Aside from the title, this book has nothing in common with the encyclopedia offered by Microsoft a few years back. This is a good reference, designed for programmers. There's coverage of the details of the file system, as well as articles on writing TSRs, device drivers and exception handlers, etc. There's a section on compatibility between DOS versions and OS/2, as well as developing for Windows.

My only problem with the book, besides it's hefty price tag of $135, is the dust jacket. The book was orchestrated and edited by friend and ex-*DDJ* columnist Ray Duncan, but somehow they forgot to put his name on the cover of the copy I received. The item they did remember is a nice bold credit for the foreword by Bill Gates. Funny how these things work out.

Tyler Sperry
editor

ARCHIVES

Ten Years Ago Today

"...Pascal with extensions is a superior language for system programming, and we believe that it is in the public interest to assist in the current effort of many people and institutions to promote wider use of Pascal in place of some of the earlier high level languages. Though Pascal may have some shortcomings for specific applications when compared to specific proprietary languages, we regard it as by far the best general purpose language now in the public domain."—*Kenneth L. Bowles "Status of the UCSD Pascal Project,"* DDJ, *March 1978.*

The object of this class is...

"Object-oriented programming began in the early 1960s with the development of Simula by the Norwegian Computation Center in Oslo, Norway. Simula was the first language to implement the Class construct. In the early 1970s, the Learning Research Group at Xerox Palo Alto Research Center began implementation of the Smalltalk programming environment. Smalltalk was the first system to be designed completely around the Class/Object concept. Many other languages have since provided support for classes, object, and subclassing, including CLU, Ada, C++ (an extended version of C), and several versions of LISP, though objects and classes are not as well integrated into these languages as they are in Smalltalk. The ease of experimentation with objects was one of the key reasons that much of the interesting user interface designs borrowed for Macintosh and Lisa were created in the Smalltalk system."—*Bruce Horn,* DDJ, *October 1985.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Turbo C vs. Quick C

Dear *DDJ*,

I read with great interest Allen Holub's article on C compilers in the October 1987 issue, in particular the comparison between Turbo C and Quick C. I largely agreed with his point of view about the programming environments, which I have little use for—they get in the way and are an additional source of compiler bugs. On other points, however, I disagree.

I finally received my Microsoft C 5.0 upgrade from 4.0 yesterday (December 4). Allen's complaints about the delays in delivery of Turbo C seem ironic to me, given the even longer delays in shipping Microsoft C and Quick C.

Allen's tests of the compilers must have been incomplete because he rates Quick C as clearly superior to Turbo C. There are several respects in which Turbo C 1.0 is superior to Quick C 1.0, and Turbo C increases its lead with Version 1.5.

First, Turbo C's documentation is much better. Second, Turbo C compiles 35 percent faster when you use Quick C's optimization switch. Third, even with Quick C's optimization turned on, executables produced by Turbo C run 15 percent faster than those generated by Quick C using my generic math function and numerical benchmark (*PC Tech Journal*, January 1988).

Fourth, and by far the worst, there is a serious bug in Quick C's numerical library: sines and tangents are apparently computed only to float accuracy, even when all real functions and variables are declared double. Alone of seven PC C compilers I have tested, Quick C fails my double-precision accuracy diagnostic on these functions. Quick C's debugging facilities are superior to the nonexistent ones for Turbo C 1.0, but Borland says it will rectify this situation "soon."

The only reason I can see for using Quick C is to migrate easily to Microsoft C 5.0. Even so, nonoptimized compilation for Quick C takes almost 50 percent of that for Microsoft C 5.0. This doesn't seem worth it to me because Quick C doesn't produce the same numbers as Microsoft C 5.0.

Users of these compilers should be warned that optimization with Quick C does not change the numerical results (though it does markedly slow compilation), but iterative multiplications and divisions produce slightly different results with Microsoft C 5.0 when you optimize the loops.

Jim Roberts
3605 Centinela Ave., #2
Los Angeles, CA 90066

Allen replies:

I disagree.

First, release dates. There are two points:

1. Borland wouldn't give me a prerelease version of Turbo C. It gave a dog and pony show down in Scotts Valley but wouldn't give me a copy of the compiler to work with at home. I'm not willing to review a product that I can't work with on a day-to-day basis, and because I had a beta version of Quick C, I felt competent to talk about it, even though it hadn't been released at that time.
2. I agree, that neither company should have advertised their products as finished until they were. The problem here is projected release dates vs. advertising deadlines. You have to place an ad several months in advance of its publication, and it is usually impossible to guess accurately when a real product will be finished. I think that both Microsoft and Borland jumped the gun, however.

Second, by my standards Turbo C's documentation is inferior to Quick C's. The two users' guides are comparable, but Turbo C's library documentation is miserable. It's poorly organized and formatted; the programming examples—if they're there at all—are trivial; and obscure and nonstandard subroutines are often documented (and I use the word loosely) in only two or three sentences. I use the run-time library documentation for my compiler on a daily basis; I read the users' guide once when I get the compiler. Consequently, the quality of the former is much more important, and Turbo C falls down in that department.

Next, I'm not sure about the comparative compile times with the newest Turbo C version. I haven't seen a copy. As I said in the review, I was using Turbo C 1.0 and a beta version of Quick C, so it's likely that things have changed. Mr. Roberts and I are testing the compilers in radically different ways, however. My benchmarks were two very large programs (versions of the Unix lex and yacc utilities) that used virtually no numeric computa-tions but did a lot of data manipulation, especially at the bit level. Mr. Roberts seems to be compiling smaller, numeric-intensive programs.

Because I disliked both of the interactive environments, I compiled with the command-line versions of both compilers, the compile process driven from a make file. I did find that it took longer to load Quick C than it did to load Turbo C. In fact, on a short program, it often took more time to load Quick C than to run it. Because I was compiling relatively large modules, however, the load time was insignificant when measured against total compile time.

Compile time is relatively insignificant in the greater scheme of things, however. If I save five hours with a good debugger but waste an hour with longer compile times, I'm still four hours ahead. Quick C's debugging facilities are indeed superior to the nonexistent ones for Turbo C, and the debugger alone makes Quick C a considerably more useful compiler, especially if you're just learning the language. Borland

(continued from page 12)

seems to be fostering rumors that it's working on a debugger, but the last time I talked to the people at the company, they wouldn't even commit to the fact that a debugger was in progress, much less give me a projected release date. It may be available "soon," but are they talking geologic time or historical time?

Other problems are reliability and portability. My Turbo C-compiled benchmark didn't run initially because of bugs in and omissions from the I/O library. There was no *signal( )* function, *getenv( )* didn't work properly (it returned a *PATH* environment when I had asked for a *PA* environment but had set *PATH* first), and so forth.

Of course, by the time you read this letter, things may have changed: Borland may have fixed the documentation, improved the library, and come out with a debugger to match dbx on a Sun workstation. But I'm only willing to discuss products that I actually have in my hands. Similarly, I test compilers by working with them rather than by running artificial benchmarks. Consequently, I'm likely to miss a few things that I don't use very much (like the accuracy of a sine function), but I think my methods give a much better feel for how a product works in general than do more formal benchmarks.

Dear *DDJ*,
We'd like to set the record straight on the shipment of Borland International's Turbo C.

Borland announced shipment of Turbo C and distributed the product on the same day: May 14, 1987. We said we'd ship in the second quarter of 1987, and we made it by over a month and a half.

We'd also like to point out that Borland collected no money before the product began shipping. As a matter of policy, Borland neither cashes checks nor bills credit cards until a product is shipped.

With the volume and depth of information that Dr. Dobbs' offers, it's hard to check every detail. Thanks for letting us contribute to your fine publication.

Borland International

The editor replies:
It was never our intention to imply that the folks at Borland were committing mail fraud, or anything of the sort. Borland has generally performed very well in customer relations.

It was incongruous, however, to have Turbo C arrive at our offices with a note from Philippe Kahn proclaiming the product was shipping "right on time"—three months after the first ads had appeared in magazines. To claim that an advertisement with a snip-out coupon and an 800 number for credit card orders isn't an official product announcement is mere sophistry; ads are designed with coupons and credit card phone numbers to generate an immediate response. Our readers had every reason to expect shipment within 30 days of the first ad.

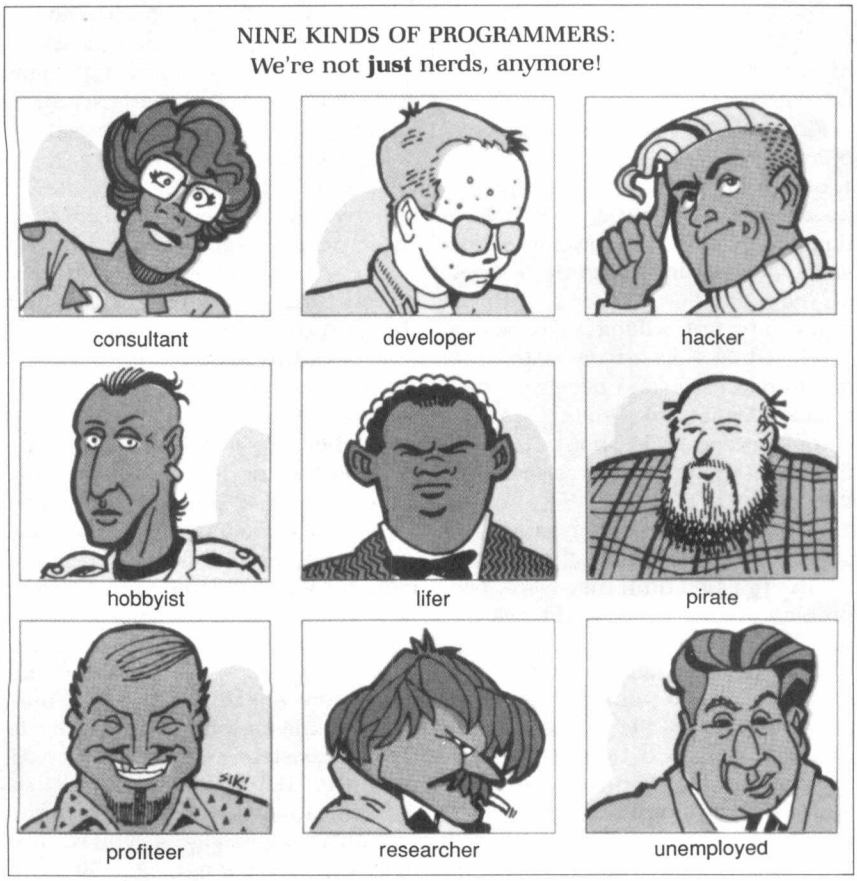
In short, if Borland had originally planned to ship Turbo C in the second quarter it would have been more honest to either postpone the ad a few months or include a disclaimer ("please allow 3 to 4 months for shipment").

Corrections

Dear *DDJ*,
Readers, your response to "An Extended IBM PC COM Port Driver" in the June 1987 issue of *DDJ* was gratifying. The version of excom.asm available from *DDJ* on diskette seems to be working, whereas exmode.c has a serious bug. Exmode will work correctly if you have two COM ports and will not do anything right if you only have one port. Specifically, in one-port systems exmode will always respond with "excom not installed" and fail to execute your commands. To correct this, change the *&&* in the first *if* statement of *main* to *||*. In addition, some early versions of exmode used *remove* as a function name. This conflicts with some libraries and can be corrected by renaming the function. How embarrassing, sorry 'bout that.

Tom Zimniewicz
Lima, NY 14485

DDJ



Object-Oriented Programming and Databases

by Jacob Stein

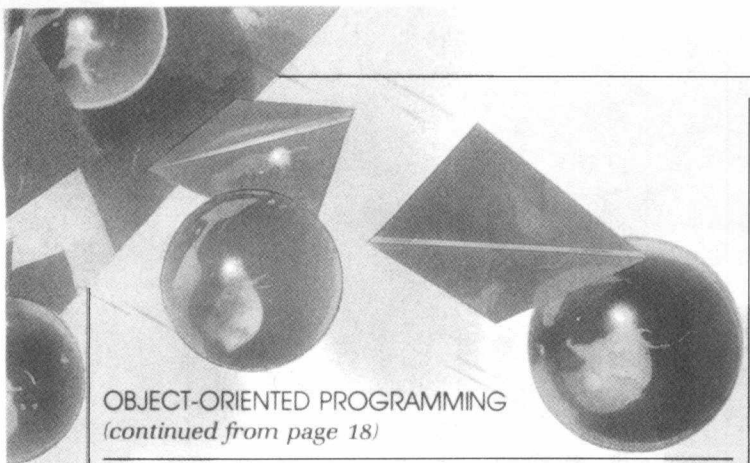
Is structured programming still relevant? In the early 1970s, E.W. Dijkstra introduced us to structured programming and Nicklaus Wirth introduced us to stepwise refinement. These programming methodologies were intended to foster a systematic and scientific approach to software development. That they were an improvement over previous methodologies, or the lack thereof, is not in doubt. What is in doubt is whether they are still germane to many applications being developed today. Structured programming appears to fall apart when applications exceed 100,000 or so lines of code. Programmer productivity fails to meet expectations and software maintenance becomes a nightmare. What we are left with is programmer fear: fear of side effects when fixing even minor bugs; fear of improving what already works, even if it does so poorly.

It may be that the complexity of the application itself is responsible for this failure. On the other hand, our understanding of the application domain may be the limiting factor. Although the underlying models for banking and other conventional applications have been understood for millenia, the underlying models for new application areas such as computer-aided software engineering (CASE) and computer-integrated manufacturing (CIM) are as much a product of the process of application development as they are inputs to the process. Structured programming assumes that the application developer stands a reasonable chance of getting it right the first time, and that once the application has been successfully coded, the application domain will not change. For many applications being addressed today, neither of these is likely.

Object-oriented programming provides a paradigm suitable for these new application areas. Instead of looking at programs as algorithms and data structures, it looks at applications as the behavior of objects and the communication between objects. This communication takes place in the form of messages to which objects respond with suitable behavior. In addition, object-oriented programming addresses the vital need to validate abstractions before committing many man-years to a detailed implementation and the fact that application domains change with time. Diedrich and Milton refer to the style of development encouraged by object-oriented languages as "fearless programming."[1]

By organizing objects into classes, which encapsulate object behavior in a manner similar to abstract data types, and organizing classes into an inheritance hierarchy, object-oriented programming may finally provide the long sought after sharing and reusability of code. The code that gives a data structure its meaning, its semantics, is no longer spread amongst disparate application programs. Perhaps one of the most frustrating aspects of trying to reuse code in a new application is that the old code almost, but not quite, fits the needs of

Dr. Jacob Stein has been a senior software engineer at Servio Logic Corp. of Beaverton, OR since 1984. He is also an adjunct assistant professor at the Oregon Graduate Center.



(continued from page 18)

the new application. In an object-oriented language, a new class can be created from an existing class by specifying only the differences between the existing and new class; all else is inherited by the new class from the existing class. PPI's belief in the encapsulation provided by object-oriented systems and lead them to trademark the term *Software-IC*.

Object-oriented languages tend to be memory-based and are geared toward supporting a single user or application. Objects either do not persist between program executions or are saved in a primitive manner that does not lend itself to sharing (for example, copying a program's image to disk). It would be difficult for a design layout tool and a design rule checker to share data, let alone run concurrently. Object-oriented data management systems rectify this situation by providing data management facilities. The trick here is to provide these facilities in a manner that provides flexibility and merges well with the object-oriented paradigm.

An Object-Oriented Model

This section outlines the OPAL programming language and GemStone data model developed by Servio Logic Corp. Most of the constructs described here are present, in one form or another, in all object-oriented languages. The concepts of objects, messages, methods, classes, and class hierarchy are essential to all object-oriented models. You should beware of systems that claim to be object-oriented yet do not contain similar concepts.

The three principal concepts of the model and language are object, message, and class. They correspond roughly to record, procedure call, and record type in conventional systems. (Table 1, page 22, gives some other correspondences.) An object is a chunk of private memory with a public interface. Objects communicate with other objects by passing messages, which are requests for the receiving object to change its state and/or return a result. The set of messages an object responds to is called its protocol (its "public interface"). An object may be inspected or changed only through its protocol. The means by which an object responds to a message is a method. A method is an OPAL procedure that is invoked when an object receives a particular message. So that each object need not carry around its own methods, objects with the same internal structure and methods are grouped into a class and are called instances of the class. The methods and structure are in a single object describing the class—the class-defining object (CDO)—and all instances of the class contain a reference to the class-defining object. Unlike some other object models, an object is an instance of exactly one class.

Every object has a unique identity. In OPAL, an object is identified by its object-oriented pointer (OOP). An object's identity is invariant: it remains the same regardless of changes in the object's state. OPAL does not support the *become:* message, which interchanges the identity of two objects. Smalltalk uses this message for screen manipulation and growing objects. GemStone's designers considered *become:* an inappropriate solution to the problems it addresses. The functionality it is used for is implemented in other ways.

Objects

Internally, most objects are divided into fields, called instance variables. Each instance variable can hold a value, which is usually a reference to another object. Figure 1, page 22, shows an *Employee* object with four instance variables: *empName*, *ssNo*, *address*, and *salary*. The *empName* instance variable holds an object that is an instance of the *PersonName* class. Not all objects are internally divided into instance variables. Certain basic types such as *SmallInteger* and *Character* are not further decomposed. These basic types are self-identifying as they have a direct representation.

The instance variables in the *Employee* object are called named instance variables. An object can have indexed instance variables, which can be viewed as instance variables with numbers instead of names. Indexed instance variables are used mainly for implementing ordered collections, such as arrays.

Messages

The basic form of all message expressions is *<receiver> <message>*. The *<receiver>* part is an identifier or expression denoting an object that receives and interprets the message. The *<message>* part gives the selector of the message and possibly arguments to the message. Every message returns a result to the sender, which is usually another object. There are three kinds of messages: unary, binary, and keyword.

Unary messages have no arguments and have selectors that are a single identifier. Assume that *emp* is a variable holding an *Employee* object. If there is a unary message *firstName* to retrieve the first name of the employee, then *emp firstName* returns a string that is the first name of *emp*.

Binary message expressions have a receiver, one argument, and a message selector that is one or two nonalphanumeric characters. To multiply 8 by 3, for example, you send to 8 the message "Multiply yourself by 3": *8 * 3*. Here, * is the binary selector for the multiplication message. Comparisons are also handled with binary messages. In the following expression:

```
(emp1 salary) <= (emp2 salary)
```

the receiver and argument of the <= binary message are both results of unary message expressions.

Keyword messages have one or more arguments and have multipart selectors composed of alphanumeric

(continued from page 20)

characters and colons. To set the third component of an array held in *anArray* to *'Ross'*, you use:

```
anArray at: 3 put: 'Ross'
```

The same effect is accomplished in other languages by:

```
anArray[3] := 'Ross'
```

The message selector here has two parts—*at:* and *put:*—because it takes two arguments: the array index and the object to be stored at that index. You refer to a message is referred to by the concatenation of its parts: *at:put*.

Methods

To construct a method you need to know what objects are visible within its scope. All the named instance variables of the receiver are available via their names. Thus, in a method for class Employee, as defined in Figure 1, the instance variables *empName*, *ssNo*, *address*, and *salary* are accessible.

A method may also have temporary variables, which are declared between vertical bars at the beginning of the method: *|temp1 temp2|*. Each user has one or more dictionaries of global variables, and those global variables can appear in methods. You need two other operators before you can write methods: the assignment operator (:=) and the operator ^, which returns the value of the expression as the result of a method.

For a unary message *wholeName* in class *PersonName* that returns the first and last name concatenated with a space between, you can use the following method:

```
wholeName
  |temp|
  temp := first.
  temp := temp + ' ' + last.
  ^ temp
```

The first statement, *temp := first*, assigns the value of the instance variable *first* in the receiver (a *PersonName* object) to the temporary variable *temp*. The statement:

```
temp := temp + ' ' + last
```

concatenates the value of temp, a blank, and the contents of the instance variable *last* of the receiver (+ is the binary message for concatenation of strings). The result of the concatenations is assigned to *temp*. Finally, the statement *^temp* returns the value of *temp* as the result of the message *wholeName*. (This particular message could be implemented with the single message expression *^ first + ' ' + last* with no need for a temporary variable.)

A method can also change the state of the receiver, by

GemStone	Conventional
object	record instance, set instance
instance variable	field, attribute
instance variable contraint	field type, domain
message	procedure call
method	procedure body
class-defining object	record type, relation scheme
class hierarchy	database scheme
class instance	record instance tuple
collection class	set, relation

Table 1: *Correspondences between concepts in GemStone and conventional systems*

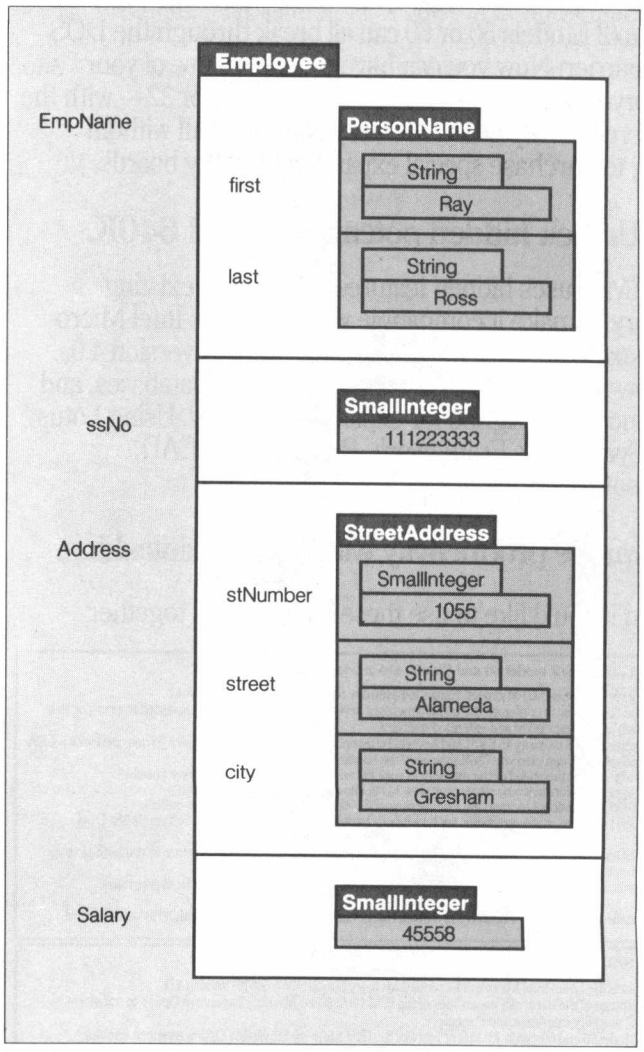


Figure 1: *An* Employee *object and its four instance variables*

(continued from page 22)

assigning new values to instance variables. Suppose objects of class *Department* have a *budget* instance variable. The following method increases a department's budget by a certain percentage, up to a limit:

```
increaseBudgetBy: aPercentage upTo: aLimit
   budget := budget + (budget * (aPercentage / 100)).
   (budget > aLimit) ifTrue: [budget := aLimit].
   ^ budget
```

The *increaseBudgetBy:upTo:* message takes two arguments, represented by variables *aPercentage* and *aLimit*. This method changes the *budget* instance variable of the *Department* object that receives the *increaseBudgetBy:upTo:* message. In the second line of the method, you see a keyword message *ifTrue:* that functions as a conditional control construct. The receiver of this message is a Boolean value. The argument of the message is a block. A block is a sequence of one or more OPAL statements within brackets and is a first-class object in GemStone. The effect of a message *aBoolean ifTrue:* aBlock is to perform the code in *aBlock* if *aBoolean* is *true*. The third line of the method returns the new value of *budget*.

OPAL includes messages for iterative control structures, using blocks with arguments. Block arguments are declared at the beginning of a block. For example:

```
[:n | (2 * n) - 1]
```

is a block with one argument, *n*. For this block to be evaluated, it needs a value for the argument. The *value:* message provides the argument and causes execution. A block returns the value of its last expression when executed. Thus:

```
[:n | (2 * n) – 1] value: 7
```

returns *13*. In general, given a value of *N*, this block returns the *N*th odd number.

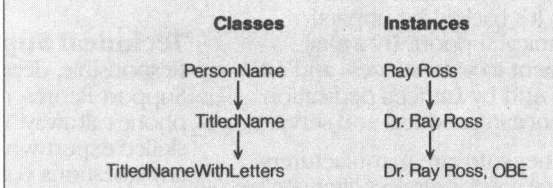
All methods you have seen so far have been defined in terms of other messages. The definitions of methods are not completely circular. At the bottom of everything are primitive methods. When the OPAL interpreter encounters a message that has a primitive method, it executes a piece of machine code rather than an OPAL method. Primitive methods exist for arithmetic, comparisons, object creation and copying, array selection, string manipulation, and set functions. The set of primitive methods in GemStone cannot be augmented by a GemStone programmer, although such an extension could be provided.

Classes

Every class is represented by a class-defining object that describes the structure and behavior of instances of the class as well as the position of the class in the class hierarchy. Any object will return the CDO for its class in response to the message *class*. Suppose the variable *assoc* holds an object of class *Association*. (*Association* is

```
Object
  Association
    SymbolAssociation
  Behavior
    Class
    Metaclass
  Boolean
  Collection
    SequenceableCollection
      Array
        InvariantArray
        Repository
      String
        InvariantString
          Symbol
    Bag
      Set
        Dictionary
          SymbolDictionary
            LanguageDictionary
        SymbolSet
        UserProfileSet
  CompiledMethod
  Magnitude
    Character
    DateTime
    Number
      Float
      Fraction
      Integer
        LargeNegativeInteger
        LargePositiveInteger
        SmallInteger
  MethodContext
    Block
      SelectionBlock
  Segment
  Stream
    PositionableStream
      ReadStream
      WriteStream
  System
  UndefinedObject
  UserProfile
```

Figure 2: *Kernel classes in the superclass hierarchy*



Class	InstanceVariables	Messages
PersonName	first last	first first: last last: fullName
TitledName	(above)+ title	(above)+ titledName
TitledNameWithLetters	(above)+ letters	(above)+ titledName (new Method)

Table 2: *Representation of people's names with titles*

OBJECT-ORIENTED PROGRAMMING
(continued from page 24)

a key-value pair used in building dictionaries.) The assignment:

```
ac := assoc class
```

causes *ac* to be assigned the CDO for class *Association*. CDOs respond to messages just as all other objects do. For example, CDOs respond to the *name* message. The result of the expression *assoc class name* is *#Association*. (Symbols are indicated with a # as a prefix.)

The instance variable names and methods for instances of a class are stored in the CDO for the class. CDOs also store the methods that instances execute in response to various messages. When an object receives a message, it consults the CDO to find out how to execute that message.

OPAL provides a class hierarchy to exploit similarities in structure and behavior of entities. (The built-in hierarchy is shown in Figure 2, page 24.) A subclass inherits structure and behavior from its superclass. Structure is inherited in that all named instance variables in the superclass are also present in any subclass. Suppose you wanted objects that represented people's names with titles. You could create a subclass *TitledName* of *PersonName*, and instances of *TitledName* would automatically have instance variables *first* and *last*. You could add an instance variable, *title*, to *TitledName* to hold the title. (See Table 2, page 24.)

A subclass inherits methods from its superclass. Thus, if *fullName* is a message for *PersonName*, instances of *TitledName* respond to that message by using the method from *PersonName*. The lookup process to determine which method corresponds to a message starts in an object's class. If the message is not defined in that class, the search proceeds in the class's superclass, the superclass of that class, and so forth. Thus, the implementation of *fullName* can be overridden in *TitledName* if desired.

A subclass can implement messages of its own. For *TitledName* you might want a message that returns a string containing the full name with title:

```
titledName
^ title + ' ' + self fullName
```

A new feature in this example is *self*, which is a special variable whose value is always the receiver of the message.

A subclass can also reimplement an inherited message. Suppose you define a class *TitledNameWithLetters* as a subclass of *TitledName*. This subclass adds an instance variable *letters* that holds the letters after a person's name, as in *'Dr. Ray Ross, OBE'*. You can reimplement the *titledName* message in *TitledNameWithLetters* to include the letters after the name.

In OPAL, unlike some object-oriented languages, instance variables may be constrained to a kind of a class. An object is a kind of class *foo* if its class is *foo* or a subclass of *foo*. If instance variable *personName* in class *Employee* is constrained to *PersonName*, then in instances of *Employee*, *personName* could be an instance of *PersonName*, *TitledName*, or *TitledNameWithLetters*. Constraints may also be placed on the elements of collection classes such as *Bag* and *Set*.

The Read-Write Boundary

In the same manner as conventional programming methodologies are not appropriate for today's applications, conventional data management systems are not appropriate for managing their data. Conventional data mangement systems are best at two things. One, they are good at batch processing large amounts of homogeneous data such as monthly credit-card billings. Two, they are good for high-transaction-rate applications such as ATM networks. The processing of data in applications such as CASE, cartography, and CIM fits neither of these two categories. The data in a cartographic database is not homogeneous, and transactions in a CASE system can take hours, or days, not just seconds.

Perhaps the best argument for integrating a data management system with an object-oriented language can be made by looking at the impedance mismatch across the interface between conventional application and database languages. Conventional application and database languages are incompatible both computationally and structurally.

Data manipulation languages use a read-write model of interaction with application code. Database calls are embedded in the application language and allow for fetching, inserting, deleting, and modifying records or tuples. Although query languages have matured, allowing for the definition of complex views through sophisticated queries, they are by no means complete, and views are difficult to update.

In any event, the communication between application and database is declarative, not procedural. The application tells the database what it wants, not how to get it. This works well when what the application wants can be specified declaritively and the database can figure out how to get it in an efficient manner. All too often, though, an application needs to intersperse several database calls amongst application code to perform what should be a single logical operation.

A similar problem arises with regard to maintaining the consistency of a database. Many database systems do not allow constraints any more sophisticated than requiring key values to be unique. The application program ends up with the responsibility of maintaining all other constraints on the data.

Consider an office management system in which several applications reserve meeting rooms. In such a system, each application would first check for the availability of the room. This might involve fetching records to check that the individual reserving the room is authorized to do so and that the room has not already been reserved. The application would then post a record to indicate the reservation and perhaps post other

records to appointment calendars. A change to the structure of the database, or the policy for reserving rooms, might require locating and modifying every application that modifies the database. In an object-oriented data management system, a message could be defined that performed all necessary checks and updates to the database. In the event that the policy for reserving a room changes, the method for reserving a room could be easily located and modified.

You might argue that in a conventional system the code for reserving a room could be factored into a single procedure—thus, achieving the same effect. In a conventional system, however, none of the applications is forced to use the procedure when making a reservation. The encapsulation provided by object-oriented data management allows the designer of a class to control access to data in instances of the class. By organizing the methods for a class along with the structural description of its instances, the impact of changes in structure are localized. This control and organization becomes more important as the size and complexity of applications grow: placing the data in one file and the code that determines its meaning into several other files no longer makes sense.

Two other negative aspects of the read-write model should be pointed out. One, a procedure may need to make numerous calls to the database. If a procedure needs to fetch 1,000 records, it may need to make 1,000 calls. In an object-oriented data management system, one message can take the place of many database operations. Two, conventional systems may be doing a lot of unnecessary copying. If a database is queried for employees who have been with a company for at least five years, the tuples corresponding to those employees are copied into the result. In an object-oriented system the result would contain only the identities of those employees.

The differences between data types supported by the application language and the data management language cause structural impedance across the read-write boundary. Although programming languages directly support arrays, most relational systems must encode them by adding a field to represent offsets. This encoding requires a translation when arrays are retrieved from, or stored to, the database. Union types and nested structures are difficult to represent using relations. Application developers are typically faced with two choices. One, they can normalize the representation so that each relation stores only one of the unioned types and only flat structures are stored. Two, they can store *uninterpreted* bit strings if they fit within the size limitations imposed by the relational system.

With the first choice, the application becomes dependent on the decomposition in order for the translation to work properly when storing and fetching. Furthermore, in the relational model, the properties of an entity must be sufficient to distinguish it from all other entities. For an employee tuple to reference a department tuple, there must be some fields in department tuples that uniquely and immutably identify departments. Using department names to identify department tuples is fine until a department's name changes. Maintaining the validity of foreign keys, such as a department name stored in an employees tuple, is known as referential integrity. Making up unique department numbers might solve the problem; however, this adds an attribute that may not be present in the world being modeled and is a burden on application developers. With the second choice, the database isn't providing any more functionality than a good file manangement system.

Suffice it to say that conventional programming languages manipulate data types by representing their structure in one place and distributing their semantics throughout the application code, whereas data managment systems, because of their limited repertoire, can only manage data structures. An object-oriented data management system combines object-oriented programming with an integrated data model to free programmers from the tedious and error-prone coding needed to accommodate a separate data management system that understands neither the execution model of the language nor the structures used by the language in representing its data types.

An Object-Oriented Data Management System

Arguing in favor of an object-oriented language that provides a common abstraction for data definition, data manipulation, and general computation is one thing; actually developing one is quite another. What follows is a discussion of object-oriented data management issues and how they are addressed by GemStone.

One of the guiding principles in GemStone's development was uniformity; all objects, be they temporaries or shared and disk-resident, should be accessed in a uniform manner. This principle helped address the issue of how objects persist. Does an application explicitly state that an object is to be persistent and must be explicitly deleted, or do objects exist as long as they are reachable from a special root collection of objects? Some folks argue that persistence through reachability is not practical for a disk-based system: garbage collection of disk-based objects incurs too much overhead.

Let's take a look at the implications of explicit deletion. In the first place, who decides when an object can safely be deleted? In a shared environment, in which many applications are using the data, this is not an easy question to answer. When an object is deleted, what is to be done with all the references to that object? The problem is extremely similar to that of referential integrity in relational databases.

Assume that it would be satisfactory to replace these references with references to a special object *Nil*. Either all the references are located and replaced in one operation, which corresponds to garbage collection, or all references are screened for deleted objects, which corresponds to incremental garbage collection. If references are screened, are all the instance variables of an object screened when the object is accessed, or are only those instance variables to which messages are sent screened? The former involves greater overhead, whereas the latter allows the propagation of references to a

deleted object: an object can be assigned to an instance variable without sending messages.

Given that explicit deletion does not incur less computational overhead, is difficult to control in a shared environment, and is less transparent to applications, persistence through reachability was chosen for GemStone. Instances of class *UserProfile* are used to represent properties of each user, including a list of dictionaries to be used in resolving symbols when compiling OPAL code for that user. Any object that is reachable from a dictionary in any user's *UserProfile* is persistent. Dictionaries are also used to manage name spaces and sharing. When an identifier is encountered and it does not correspond to a temporary, instance, or class variable, the dictionaries are searched to find an object corresponding to that identifier. A user may have any number of dictionaries to accommodate various degrees of sharing.

Data Integrity

Three kinds of failure can compromise the integrity of a database: an application may fail to complete because of a run-time error; the processor managing secondary storage may fail; and the media used to store objects may fail. Protection against media failure is achieved by replicating objects on disk.

Failures of the first two kinds require the careful posting of an application's changes to the database. By ensuring that either all, or none, of an applications's changes are posted, inconsistent states of the database are avoided. This all-or-nothing form of posting changes is known as atomicity. Applications generally use *commit* and *abort* commands to define the changes that are to be atomically posted. When an application commits, an attempt is made to post all changes made by the application since the last commit or abort (you pretend that the first thing an application does is an abort). If all changes are successfully posted, then the commit succeeds; else the commit fails and the application is so informed. When an application aborts, all the changes made since the last abort or successful commit are thrown away.

The activity that occurs between an abort or commit and the succeeding abort or commit is known as a transaction. When a failure of the first two kinds occurs, it is treated as though the transaction had aborted. When the processor fails, a lot of fancy *footwork* is required to guarantee that all the needed information is safely on disk, but it works.

The two basic choices for implementing atomicity are logging and shadowing. With logging changes are posted directly to the database and logged. When a transaction aborts, the log is used to restore the database to its previous state. A successful commit allows the log to be thrown away and a new log started. With shadowing, changes to an object are posted against a copy (a shadow) of the object. When a transaction aborts, the shadow copies of objects are thrown away. The database is left alone, as it has not been modified. For a transaction to successfully commit, all modified objects must be carefully replaced with their shadow copy.

Concurrency Control

Any system that allows concurrent access to shared data must handle conflicting changes made by applications running concurrently. Either conflicting changes are prevented (pessimistic concurrency control), or conflicting changes are discarded (optimistic concurrency control). Pessimistic concurrency control requires that a transaction state its intention prior to accessing an object by acquiring either a read-lock or a write-lock. While a transaction holds a lock on an object, other transactions are prohibited from acquiring a lock that would allow conflict. Optimistic concurrency control allows a transaction to proceed as though it were running alone. At transaction commit, the changes made by the transaction are checked for conflict with other transactions. If a conflict is detected, the changes made by the transaction must be undone.

As you may have noticed, concurrency control and data integrity are strongly related. Pessimistic concurrency control is generally bundled with logging, optimistic concurrency control with shadowing. Conventional data management systems tend to use locking and logging. The advantage of optimism and shadowing is that transactions do not manage locks. The down side is that arbitrary amounts of work can be lost when conflict is detected at commit. The situation is reversed for locking and logging. One further disadvantage of locking is deadlock: two transactions are unable to complete until each gets a hold of a lock the other is holding.

For the initial implementation, optimism and shadowing were chosen. This choice was guided by a desire for uniformity. As GemStone supports general programming

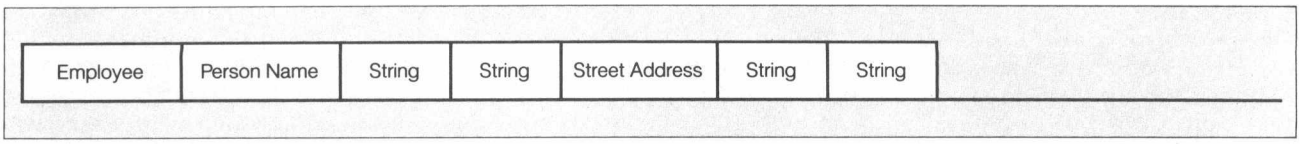


Figure 3: *Clustering of an Employee object*

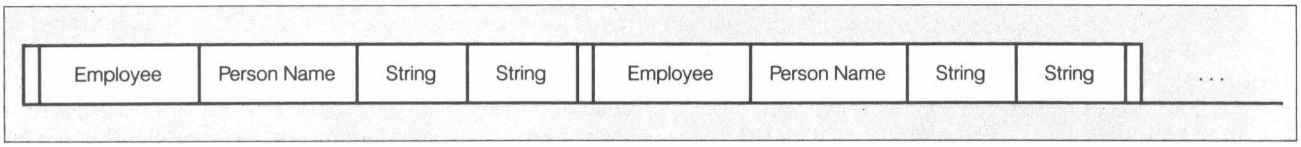


Figure 4: *Clustering of Employee objects omitting their addresses*

and data *manipulation* in a single language, the objects in a GemStone database span the continuum between objects that correspond to values of variables in conventional programming languages and those that correspond to large design objects as may be found in VLSI databases. Conflict is unlikely on objects that correspond to program variables. For one thing, they are unlikely to be shared or persistent. Imposing pessimistic concurrency control on objects at this end of the continuum is an unnecessary burden.

Optimism and shadowing allow GemStone to provide each transaction with a private workspace, within which the activities of other transactions cannot affect the transaction's progress. At commit, privacy is abandoned, changes in the private workspace are made available to other transactions, and changes committed by other transactions become visible. Servio Logic Corp. realized, however, that in the case of large design objects, the amount of work that must be undone when conflict is detected may be unacceptable. They therefore began investigating supporting pessimism control in addition to optimism.[2] The goal was to make pessimism optional and, to as large a degree as possible, not of concern to applications that chose not to use it. Support for pessimism will be included in a forthcoming release.

Concurrency control is one of the areas in which object-oriented data management offers great promise. Recent research efforts in programming languages have explored the notion of behavior-based concurrency control. As a simple example, consider a savings account to which deposits are credited and withdrawals are debited. There is no intrinsic reason why two transactions that are both crediting a given account need conflict. Neither transaction is concerned with the final balance after the credit. All they care about is that the proper amount is credited to the account; the order in which the amounts are credited is not of concern. Basically, each transaction adds the amount deposited to a list of credits to the account. The credits are processed in the order in which they were added to the list. That the credits were added to the list in the opposite order to that in which the two transactions committed has no effect upon the desired behavior. The read-write barrier of conventional systems prevents this form of concurrency as the database has no knowledge of the semantics of operations beyond read and write.

Large Objects and Large-Object Space

Although relational systems can support a large number of tuples, they generally do not allow tuples to be larger than a page. Object-oriented systems must support large objects as well as a large numbers of objects. If large objects are not supported, application developers will have to encode large objects into objects no larger than those supported by the system. A GemStone system supports 2^{31} objects (2^{32} counting instances of *SmallInteger*), and an object can contain 2^{31} instance variables.

When an object is, or grows, larger than a page, it is broken into pieces and is no longer stored contiguously. Large objects can be accessed and updated without bringing the entire objects into memory. Large objects can also grow and shrink without copying the entire object.

The basic data formats provided by an object-oriented system must support reasonably direct and efficient implementations of user-defined classes. Although relational systems support both records (tuples) and sets (relations), arrays must be encoded. A data management system must support all three.

The underlying basic storage formats must in turn efficiently support the basic data formats. GemStone supports five basic storage formats—self-identifying (for example, *SmallInteger*, *Character*, byte (for example, *String*), named, indexed, and nonsequenceable collections. The byte format is used for classes whose instances may be considered unstructured. Structure is imposed by the methods that operate on the objects. The indexed format supports access to components of an object by integers, as in instances of *Array*. The byte and indexed formats support, without copying changes in the size of an object. The named format supports access by instance variable names. Classes whose instances have both named and indexed instance variables are supported by a hybrid format. The nonsequenceable collection (NSC) format supports collection classes such as *Bag* and *Set*. The members of such collections are not identified by name or index; instead, collections can be have members added, removed, or enumerated, and efficienlly support union, intersection, difference and tests for membership.

Two other areas that need to be addressed by data management systems are clustering and associative access. Clustering is the placement of objects that tend to be accessed together near each other on the disk. The objects are placed on as few, preferably contiguous, pages as possible. By so placing the objects on disk, fewer disk accesses are required to bring these objects into memory.

Consider the *Employee* object of Figure 1. If the *clusterDepthFirst* message were sent to the object, its layout on disk would be as in Figure 3, page 30. Since the *ssNo*, *stNumber* and *Salary* instance variables are selfidentifying, they are directly represented in their containing objects and need not be clustered. Now consider a collection of *Employee* objects. If enumeration of the elements of the collection occurs frequently, the elements can be clustered together. If these enumerations tend not to access the addresses of employees, it may be desirable to omit addresses from the clustering. The layout on disk of the employees would be as in Figure 4, page 30. By omitting addresses from the clustering, employees and their frequently accessed instance variables can be enumerated with even fewer disk accesses. OPAL's clustering protocol is flexible enough to allow such clustering.

An associative access is a search of a collection based upon the internal state of the collection's elements—for example, a collection of employees can be associatively accessed for employees who live on a certain street. Even with clustering, searching large collections by a sequential scan may yield unacceptable performance.

(continued from page 33)

Associative accesses should take time that is no more than logarithmic on the size of the collection. Hashing would allow the associative access of an employee with a particular social security number in nearly constant time, regardless of the size of the collection being searched. B-tree indexes support associative access in time that is logarithmic on the size of the collection and efficiently support range queries, such as locating those employees whose salary is within a given range.

There are many issues with regard to associative access in object-oriented systems.[3] For example, should indexes index all instances of a class or only instances of explicit collections? How is the use of indexes to be indicated? Should indexes be based upon the structure or protocol of objects? If by structure, should indexes be on identity or value? GemStone supports B-tree indexes into explicit collections using the structure of objects. Both identity- and value-based indexing is supported. A limited calculus sublanguage is provided to make use of indexes. The language was constructed so that associative queries may be viewed as procedural OPAL code.

State of the Technology

Object-oriented languages and data managment are emerging technologies. The first commercially available data management systems have only recently arrived on the scene. Deciding when and if to jump on the bandwagon is difficult. Strong interest in object-oriented systems, as indicated by the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA) becoming ACM's third largest conference in just two years, is not sufficient. Other promising technologies of the past have failed to yield their expected benefits.

If, however, you are encountering the frustration with structured programming I discussed in the introduction, you might just browse through the proceedings of the OOPSLA conferences[4,5] to see what advances are being made and the kinds of applications being developed using object-oriented systems. You may well be surprised at the significant applications being developed with object-oriented technology. GemStone, for example, is being used by an agency of the U.S. government to develop a new, automated coastal chart production and maintenance system. In this system, GemStone manages both the static feature information and the procedural knowledge to render that information into readable and reliable charts. At another U.S. government facility, dedicated to research in CIM, GemStone is being used to manage information flow between the many, often incompatible, systems involved in computer-integrated manufacturing.

You may find that object-oriented technology is more mature than you thought. You might start asking hard questions about performance and how to compare object-oriented systems. Many argue that part of the price to be paid for the benefits of object-oriented systems is an apparent increase in the consumption of machine resources. It may well be the case, however, that many applications can be developed using today's technology and still yield adequate performance. After all, an application that can be developed quickly, that is easily maintained, and whose performance is adequate is often preferable to missed production deadlines, buggy code, and blazing performance.

In the same manner as relational systems have markedly improved their performance over the last decade, so will object-oriented systems. Hardware will continue to get faster and cheaper. If performance isn't quite adequate on the hardware you are running today, it will be tomorrow.

Convinced? Want one? Which one? That may be a hard decision. For one thing, there is no single object model. In some models the collection of all instances of a class is meaningful; in others it isn't. Some models support explicit deletion; others don't. There are object-oriented extensions to C. Will these C extensions perform better than systems whose heritage is Smalltalk? If so, there may be a price, such as the slower development and more difficult maintenance implied by a compile-and-link methodology. Perhaps what you really need is the EXODUS extensible database system being developed at the University of Wisconsin or the POSTGRES system being developed by Stonebraker and crew at Berkeley.

What about benchmarks? A good set of benchmarks would make comparison shopping easier. There are problems too numerous to mention here. It's difficult enough developing benchmarks for relational systems or any given programming language. It's even harder for systems that integrate different language extensions with different object models.

See you at OOPSLA.

This article was condensed by the author from Development and Implementation of an Object-Oriented DBMS *by David Maier and Jacob Stein.*

Notes

1. J. Diedrich and J. Milton, "Experimental Prototyping in Smalltalk," *IEEE Software*, vol. 4, no. 3 (May 1987).

2. D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," *Proc. International Workshop on Object-Oriented Database Systems* (Asilomar, Calif.: IEEE Computer Society Press, September 1986).

3. D. J. Penney, J. Stein, and D. Maier, "Is the Disk Half Full or Half Empty?: Combining Optimistic and Pessimistic Concurrency Mechanisms in a Shared, Persistent Object Base," *Proc. Workshop on Persistent Object Stores* (Appin, Scotland, August 1987).

4. *Proc. of ACM Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA-86)* (Portland, Oreg., October 1986.) Also published as *ACM SIGPLAN Notices* vol. 21, no. 11 (November 1986).

5. *Proc. of ACM Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA-87)* (Orlando, Fla., October 1987). Also published as *ACM SIGPLAN Notices*, vol. 22, no. 12 (December 1987).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

An Italic Font in C for the EGA and VGA

A custom font for the EGA in Turbo C with source code for a resident program that makes the font permanent

by Andrew J. Chalk

Innovative screen displays sell software, which is one reason why so many developers provide demonstration disks, often free. One thing that makes a program distinctive is having a custom typeface for the text that it displays. Until the advent of the EGA, this was a luxury reserved for programs that operated in graphics mode.

Developing a nongraphics application to operate in graphics mode (for example, Framework in its EGA two-color graphics mode) is more costly than using the strong text mode support in the PC, however. Furthermore, prior to the EGA, graphics (with the exception of Hercules) were too poor to be very attractive. The EGA was such a rarity a year after its introduction that Peter Norton could write in his authoritative 1985 *Programmers's Guide to the IBM PC* that "we won't be discussing the 64-color palette of the EGA/ECD combo because it's quite rare and specialized and doesn't really fit into the mainstream of the PC family" (page 77).

This situation changed with the advent of EGA clones from Chips and Technologies and others. At the present time, a no-frills EGA card sells for around $150 and will probably cost around $100 before the end of 1988. At least for a while, the EGA will be the de facto video standard, so it behooves developers to take advantage of its special features.

In this article I show how to exploit one of the interesting features of the EGA—the ability to replace the standard character font in text mode with one of your own choosing. As an example, I use an italic font and provide source code for a TSR that loads the font into RAM so that DOS and (most) application programs can use it. The TSR is necessary because the EGA reloads the ROM character set when the video mode is changed. By intercepting the BIOS for video mode changes, you can reload your custom font in its place. The TSR can be deinstalled, freeing up the 17K of memory it occupies for other programs and preventing incompatibilities.

The source code is written in C (Borland International's Turbo C, to be precise), so this article also serves a second purpose—explaining some of the techniques for writing resident code in high-level languages. As I will explain later, I have concluded that this is basically a bad idea. If the on-line services and bulletin boards are good indicators, however, it is a subject of great interest, so it is worth spreading the techniques just so that others can become similarly disabused of the practice.

In order to use the code included here, you first need to understand fonts on the EGA. Then I'll talk about the C implementation in the source code. After that I'll discuss the TSR aspects of the source code and why you should write TSRs in assembly language.

Andrew J. Chalk is president of Magna Carta Software, P.O. Box 475594, Garland, TX 75047.

Fonts on the EGA

Programmers writing for the monochrome display adapter (MDA) and the color graphics adapter (CGA) are stuck with the character sets provided in those board's ROMs. If you want a different character set, say as users of APL do, you have to replace the ROM. On the EGA, things are different. The fonts are "soft," meaning that although the ROM character generator is used by default, it can be replaced by a character set of your choosing. In fact, the EGA can support four character sets in what IBM calls four different blocks. Normally, you use block 0.

The EGA has BIOS support for the loading of an alternate character set

through interrupt 10h, function 11h, subfunction 0. You make a call to this function with *ES:BP* pointing to a table containing your font (in a format I will explain), *DX* set to the ASCII ordinality of the first character in your character set, *CX* set to the number of characters in your character set (maximum 100h), *BH* set to the number of bytes per character, and *BL* set to the block to load (usually 0). These parameters provide the BIOS with sufficient information to load your font because of the way that fonts are stored.

Figure 1, below, shows a letter *g* as a magnified version of its screen image and as a stored character in memory. The figure assumes an enhanced color display in 25-line mode, in which case each character is 14 scan lines high and 8 pixels wide. A 25-line screen therefore fills 14×25, or 350 scan lines, as does the EGA. On the monochrome display, a 14×9-character box is used, and on the regular color display, the CGA 8×8-character box is used. The 43-line mode that is popular on the EGA driving an enhanced color or a monochrome display is achieved by loading the 8×8 ROM character set (because 8×43 is 344, just less than the 350 scan lines available). BIOS support exists for this, too, although two bugs in the original EGA BIOS make its implementation too big a subject to digress into here.

Let's assume that the EGA is driving an enhanced color display. In this case, each character is 14 scan lines high and 8 pixels wide. Representing this in RAM is simplified by the fact that each pixel that forms part of a character can be considered to be either "on" or "off" when a given character is on the screen. This means that the state of each pixel can be represented in binary by 1 bit. Furthermore, the designers of the EGA seem to have chosen a character width of 8 because this permits each character-scan line to be represented by exactly 1 byte.

In Figure 1, the hexadecimal values of each scan line are shown above the character. The arrows that lead in the direction of memory show that the letter *g* is stored as 14 contiguous bytes of data. Because *g* has an ASCII value of 67h, the characters next to it are the ASCII values 66h and 68h. The latter of these is *h*.

When you load a custom font into the EGA, you tell the BIOS, through *ES:BP*, the address of a buffer containing your chosen characters. If you want to load a complete character set, this buffer is 3,584 (14×256) bytes long. The EGA lets you load fewer than 256 characters, and it lets you choose the starting position in the ASCII sequence. The sequence of characters for any one load operation must be consecutive members of the ASCII character set, however, or you will get garbage on the screen.

Note two important limitations of the EGA font features. First, characters have a fixed width (but may vary from 1 to 32 scan lines high), so you do not have the same flexibility as in graphics modes. Second, although your 14×8 fonts work fine on a monochrome monitor in 25-line mode, a different character set is required if you support different numbers of screen lines. Suppose, for example, you were going to incorporate a feature into a database such that the user could press a hot key and immediately switch into 43-line mode and thereby see more records in "table view." If you had a custom typeface, you would need a 43-line-mode equivalent of it that would be loaded at the same time.

Notwithstanding these limitations, the custom font capability of the EGA is impressive. As I stated earlier, a video mode reset restores the ROM character set, so your application should perform a reload operation every time it performs a video mode reset. Furthermore, if you want to load your custom fonts only in certain video modes, you should check the video mode before loading. The example program ITALIC.C in Listing One, page 50, only loads the custom character set in modes 0-3 (text modes) and 7 (monochrome).

A Resident Italic Font

The example program consists of

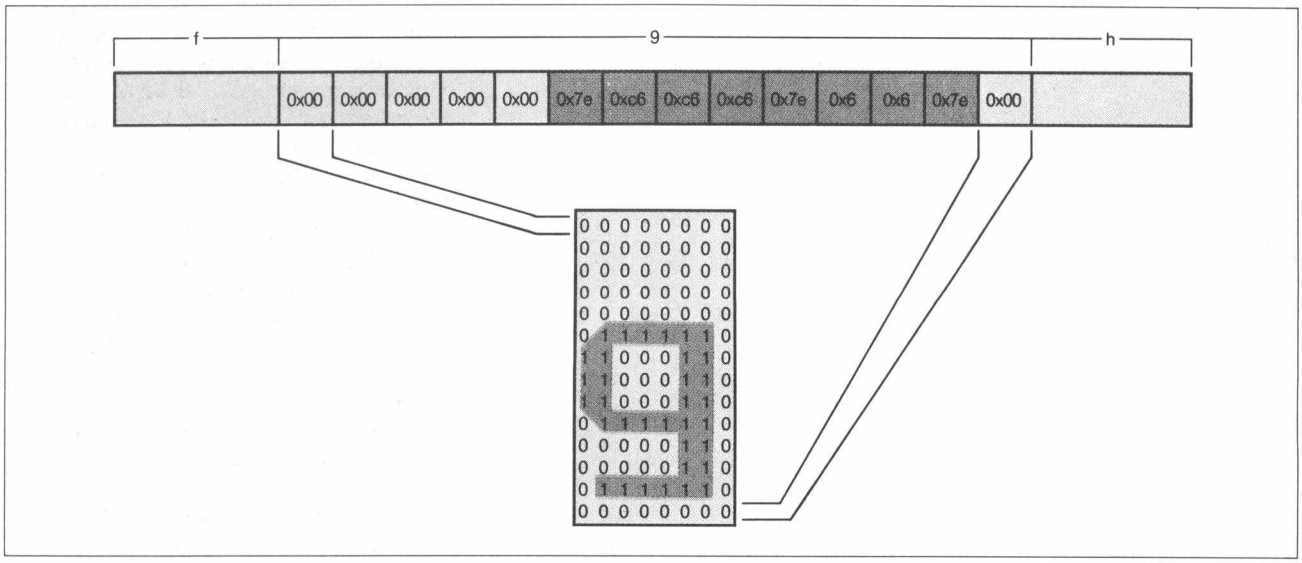


Figure 1: *EGA character representation on the screen and in memory*

ITALIC FONT IN C

(continued from page 37)

two parts: ITALIC.C is the source code, and ITALIC.ASC (Listing Two, page 52) contains the code for the font in a form I will explain shortly.

The program first checks the system configuration to see if custom fonts are supported. This is done by means of the function *get_video_info( )*, which first checks for the presence of an EGA in the system through the recommended BIOS call (function 12h). If an EGA is not present, the value of *BL* is returned unchanged, in which case you exit with a message explaining to the user that an EGA is required. There is a practice in the literature of checking for the IBM signature in the EGA BIOS. Not only is this kludgy but it is also specifically disapproved of in the EGA BIOS listing.

It is not enough to know that an EGA is present—it must also be active. The user may, for example, have an EGA driving a color monitor and an MDA driving a monochrome monitor and be using the MDA. To see if the EGA is active, you check that bit 4 of the EGA information byte in the ROM data area at *0:0487h* is 0. If not, you exit with a message to the user explaining that the EGA must be active.

If space had permitted, I would have included code to save the video configuration and switch adapters. The saved information could then be used to restore the ex ante state of the machine on exit. For the same reason, I have also omitted code to detect a switch of adapters while ITALIC is resident. Be warned: ITALIC illustrates a technique; it is not a full-blown professional program.

Having determined that an EGA is active, you then determine whether it drives a monochrome or a color monitor. The value of *BH* is 1 if monochrome and 0 if color. The result is used to set the global variable *ega_color* appropriately. If a color monitor is in use, you test for an enhanced color display. If it's not present, you exit with an explanatory message because the EGA will use the 8×8 font on a regular color display and my example requires 14 scan lines per character.

If the system checks out, you return to *main( )* and set the video mode based on the type of monitor in use. You then check whether a copy of the program has already been installed. This involves scanning down through memory from the program segment prefix (PSP) for two identification words inserted in the global data area near the top of the program. These words are *our_id1 = FACh* and *our_id2 = 1000h*. The function *already_installed( )* first peeks at the offset of *our_id1* with the segment value equal to 1 less than the PSP. The segment value is then decremented until 0 is reached or a match is found. If a match is found, the next word is peeked at and compared with *our_id2*. If you assume that all characters are equally likely, the chance of erroneously concluding that ITALIC is resident when it is not if you load halfway up memory on a 640K machine is 1 in 13,158. If you are uncomfortable with this, you can lengthen the odds by searching for more than two words.

If you find a copy of ITALIC, you deinstall it and print a message telling the user. The mechanics of deinstalling a resident program are explained in the next section. If ITALIC is not present in memory, you can proceed with installation. There are three distinct phases of this process.

First, you only want to replace the alphanumeric characters in the ASCII set. Box-drawing characters simply don't draw boxes if they are italicized. My strategy is to load a copy of the whole 14×8 ROM character set into the buffer *fontarray* and then load only the alphanumeric characters from the data file ITALIC.ASC. The advantage of this is that your .EXE file need not be swollen with unnecessary data for characters that do not differ from the ROM 14×8-character set. Retrieving the ROM character set is easy thanks to BIOS support and is accomplished in *get_egafont( )*. Next, a *for( )* loop overlays the ASCII characters 32 through 127 with italic characters.

If you refer to the listing of ITALIC.ASC, you find that the italic font data is set up as a two-dimensional array with each row consist-

ing of the appropriate hexadecimal values for a single character. The listing shown is actually the output of FONTEDIT, a full-screen, real-time EGA font editor included in C Windows Toolkit (see the "Availability" section at the end of this article). Don't worry about typing the listing; download details are given at the end of the article.

The second stage is the loading of your font (telling the EGA to use it) using interrupt 10h, function 11h, as described earlier. This is accomplished by the function *load_user_egaxfont( )*. At this point the screen display immediately changes to reflect the new font.

The third stage is the resident installation of a replacement interrupt handler for interrupt 10h so that you can detect video mode changes and reload your font if necessary. First, you save the PSP and environment pointer in the PSP at offset 2ch to use for later deinstallation. Next, you save the old interrupt 10h address using *getvect( )* and install your own handler with *setvect( )*. Finally, you terminate and stay resident (more on this in the next section).

The new font affects every character on the screen. The italic font presented here is probably not distinct enough for serious text work, but it could be edited to be so. The italic font in Microsoft Word is created through exactly the same kind of techniques. Other fonts along the lines of the large selection supplied with the Hercules Graphics Card Plus that are practical fonts for text-intensive work are also possible.

Programming Resident Programs in High-Level Languages

As I stated in the introduction, the experience of writing this (simple) TSR in C has lead me to conclude that such programs are best written in assembly language. There are several reasons for this. Perhaps the most important one is the sheer size. ITALIC occupies 13K RAM, of which 3,585 bytes are the font buffer, 1,344 bytes are the replacement font data, less than 100 bytes are for other global data, and 512 bytes are for the .EXE header. The remaining 11,500 odd bytes are "code." It is this portion that assembly language could shrink down to perhaps 3,000 bytes (I have not done it for this program). A second reason in favor of assembly language is the irrelevance of the portability issue.

An argument often legitimately raised in favor of high-level languages is their advantage in terms of development time. In fact, for resident code, even given that I was working without the benefit of a large literature on the ins and outs of programming TSRs in high-level languages such as exists for assembly language, so many problems resulted from having a compiler between me and the machine that I spent a lot of time inside the debugger. Assembly-language TSRs are easier to debug.

It can hardly be argued that the problems arose from the choice of language.

It can hardly be argued that the problems arose from the choice of language. Wasn't it the low-level links that made C so popular as an application language? It is also difficult to argue that the choice of compiler was the problem. Turbo C has library support for all the function calls associated with resident code. Although it lacks a built-in debugger at the time of writing, Periscope does an admirable job. In-line assembly language is also available, but the objective here was to not use a single line of in-line code (that isn't really writing a TSR in a high-level language).

It might be argued that programming TSRs obviates the need to learn assembly language. The last person I would recommend to write a TSR in a high-level language is someone who did not know assem-

bly language. Ironically, the alleged portability of C code to different compilers (because it is a high-level language) becomes the opposite with TSRs. Because the generated code differs across compilers, code that runs flawlessly compiled with one compiler can produce subtle and hard-to-track bugs on another. The truly "safe" code becomes the one that is "immune" from portability virtues—assembly language.

These objections aside, here is how you implement a simple resident program in Turbo C. Bear in mind that this program does not access the disk or the keyboard, so many TSR techniques are not discussed. For a fuller treatment I recommend *Turbo C: The Art of Advanced Program Design, Optimization and Debugging* by Stephen Randy Davis. I have no doubt that ITALIC could be implemented more efficiently in C than I have done here, but the relevant question is how these improvements compare with the real alternative—assembly language.

First, consider the way that you would implement this TSR in assembly language. Near the top of the source code would be the resident section, preceded by a jump to the installation section, which you would jettison when the resident part was installed. In C you perform the same installation steps of saving the old interrupt 10h vector and installing your own.

A problem arises when you wish to terminate and stay resident. Turbo C contains the *keep( )* function, which implements interrupt 21h, function 31h, and requires the number of paragraphs of memory to reserve as one of its parameters. Whereas this is simple to compute in assembly language, it is not so in high-level languages generally or in Turbo C in particular (the Turbo C manual does not even mention this problem).

The method I used in ITALIC was discovered by Dean McCrory and generously posted by him on CompuServe. Turbo C uses two internal variables: *__psp* (to contain the PSP address) and *___brklvl* (to contain the address of the end of the initialized and uninitialized data). As memory is dynamically obtained and released, *___brklvl* is adjusted accordingly. If you visualize Turbo C memory allocation as in the diagram for the small model in the *User's Guide* (that is, the data segment follows the code), then adding *___brklvl* to *DS* and subtracting the PSP address gives the size of the code and data. That is what I do in the *keep( )* statement at the end of main.

Although this is ingenious, you should be aware of some potential problems and limitations. First, this will not work in the large data models (compact, large, and huge). Second, I do not know what happens, but you must presumably not *farmalloc( )* any memory you wish the resident program to use. Third, this is undocumented and should be considered not fully tested.

The other part of the TSR code is the deinstallation routine, including the deallocation of the program's memory. This practice appears to be little known in both C and assembly language (it is certainly not officially documented). It probably enhances the value of a TSR to the user if it is removable, however.

The first thing you do is restore interrupt 10h to its original value. Next, you must deallocate memory. Deallocating the space occupied by a resident program is as easy in C as in assembly language, and the technique is the same. You must remove the program itself and its copy of the environment. In order to do so, when you first load the TSR, you must store the PSP and the contents of the environment pointer, which is located at offset 2ch in the program's PSP.

When you try to install ITALIC, *already_installed( )* finds a match and stores the data segment address of the installed copy in the global variable *old_ds*. When you deinstall the program, you peek at *old_ds*, offset by the address of *old_psp* to get the PSP, and then you repeat this using the offset of *old_env* to get the environment address. Next, you deallocate the program memory using interrupt 21h, function 49h. *ES* must hold the address of the installed program's PSP. Finally, you repeat this function call with *ES* pointing to the installed program's copy of the environment.

It is instructive to run CHKDSK before and after installation to confirm that this procedure works. I have found it to be reliable, but TSRs are a world without rules.

Summary

The techniques for loading custom fonts described here give a program an edge of distinctiveness in an ever more crowded software marketplace. Not only is this now worthwhile for developers because of the greater abundance of EGAs but the code also works on the VGA. This means a fairly long time span for the investment in program development to pay off. Graphics environments offer users custom fonts, but program development is longer. The custom font facilities of the EGA offer a faster development path that does not require the wholesale recoding of applications.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

C Windows Toolkit, is a programmers' windowing and video-handling toolkit for Borland International's Turbo C, Mix Power C, and Microsoft C, Versions 4.0, 5.0, and Quick C. C Windows Toolkit offers strong EGA support, including FONTEDIT, an EGA font editor. It is available from Magna Carta Software.

DDJ

(Listings begin on page 50.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Threaded Binary Trees

Ease of traversal is the major advantage of a threaded binary tree.

by James Mathews

I recently discovered a data structure called a threaded binary tree that combines the quick random lookup qualities of a binary tree with the easy traversal of a linked list. This data structure provided a clean and reasonably elegant solution to an application program I was developing. In this article I describe how to create and traverse threaded binary trees and provide a set of functions written in Microsoft C that illustrate how I implemented this data structure in my application.

I'm the type of programmer who likes to tinker with programs just to see how they work. That particular personality trait recently lead me to work on an interactive spelling checker for the letters and documents I write (it certainly would have been more cost-effective simply to buy a spelling checker).

The spelling checker required a data structure that would allow all the unique words in a document to be identified and then retrieved in sorted order for comparison with the dictionary. I knew that a binary tree would be a fairly efficient way of identifying and sorting the unique words in a single step, and methods of building and traversing binary trees are well known and documented.

However, I also wanted the ability to scan back and forth interactively in the list of words not found in the dictionary and select whether a particular word was to be added to the dictionary, marked as misspelt, corrected interactively, or simply ignored. And here is where I ran into trouble with a binary tree—all my solutions to moving back and forth interactively to selected nodes in the tree involved more code than seemed necessary or reasonable. At one point I even considered converting the binary tree to a doubly linked list once the (possibly) misspelled words were identified, just to facilitate scanning back and forth over the words.

And then, while browsing though *Fundamentals of Data Structures* by Horowitz and Sahni,[1] I came across a description of threaded binary trees. Great, a threaded binary tree seemed to be just what I needed.

Each node of a binary tree typically contains pointers to left and right child nodes. If a particular node does not have a left or right child, the corresponding pointer has a null value. In fact, it turns out that slightly more than half of the pointers in any binary tree will be null. For a binary tree with N nodes, there are 2N pointers (two per node) and N+1 of those pointers will be null (refer to Horowitz and Sahni or another text for a discussion about why N+1 pointers are null).

Threaded binary trees differ from other binary trees in that the null pointers to nonexistent left and right child nodes are used as "threads" to point to prior and successor nodes in the tree. Using the (otherwise) null pointers as threads to prior and successor nodes allows a threaded binary tree to be traversed in order almost as easily as a linked list while retaining the quick lookup of a binary tree. Figure 1, page 43, illustrates (a) a standard binary tree and (b) the same tree with the null pointers replaced by threads.

The Code

Listing One, page 58, shows a C language header file (tbtree.h) that defines a *TREE_NODE* structure containing some of the fields used by my spelling checker application. A node is created in the threaded binary tree for every unique word found in the document being checked. Each *TREE_NODE* contains a pointer to the node's word

James Mathews, Blue Sky Software, P.O. Box 232, Absecon, NJ 08201.

(stored as a null-terminated string), an integer used to hold flag values, a usage count indicating how many times the word appears in the document, and pointers to left and right child nodes.

When traversing a threaded tree structure, it is necessary to know if a particular pointer field contains the address of a child node or if the pointer field contains a thread to a prior or successor node. The tbtree.h header file also defines two flags, *RBIT* and *LBIT*, which indicate if the right and left child pointers contain valid child node addresses or if they contain threads to other nodes. If the *RBIT* flag is set in a node, then that node has a right child node; if *RBIT* is not set, the right child pointer is a thread. Similarly, the *LBIT* flag identifies the usage of the left child pointer.

Notice that the *RBIT* and *LBIT* flags are the only extra storage required in the tree nodes. The threads occupy the right and left child pointers that would otherwise have been left null. Knuth[2] suggests that even these flags could be eliminated if a child node always resides at a higher memory address than its parent—a child node would have a higher address than the current node, whereas a thread would always contain a lower address. In my application, I chose to play it safe and added the two 1-bit flags. Because I already required a group of other flags for each node, these additional flags did not increase the size of a node.

tbtree.c in Listing Two, page 58, shows the routines I created to build and traverse a threaded binary tree. The function *add2tree( )* adds a new word to the tree each time it's called provided the word doesn't already exist in the tree. *add2tree( )* compares the prospective new word to existing nodes in the tree until it either locates a node already containing that word or it locates a node to which the new word should be added as a right or left child. If the word already appears in the tree, *add2word( )* simply increments the word's usage count and exits. If the word needs to be added to the tree, *add2tree( )* invokes the *linsert( )* or *rinsert( )* function to add a node as a left or right child, respectively.

Functions *linsert( )* and *rinsert( )* implement the logic required to add a node to a threaded binary tree. Notice that these routines are actually quite simple. To add a new left child node, the new node is given the left child pointer from the parent node, the new node's right child pointer becomes a thread back to the parent, and the new node is linked to the parent as a left child. Function *rinsert( )* is an equally simple implementation to add a right child to a node.

I should point out that the *linsert( )* and *rinsert( )* routines call function *talloc( )* to allocate the memory for a new node. The first implementation of these routines called the standard *malloc( )* library routine to allocate memory, but in analyzing the performance of the program on large documents, I found that *malloc( )* was by far the largest consumer of CPU time. I therefore created the *talloc( )* function to allocate tree nodes more efficiently. For relatively small tree structures, *malloc( )* could certainly be used without noticeable degradation.

The functions shown in Listing Two require the presence of a specially initialized root node (although its structure is the same as that of any other node). The *add2tree( )*, *linsert( )*, and *rinsert( )* functions build the threaded binary tree as a left

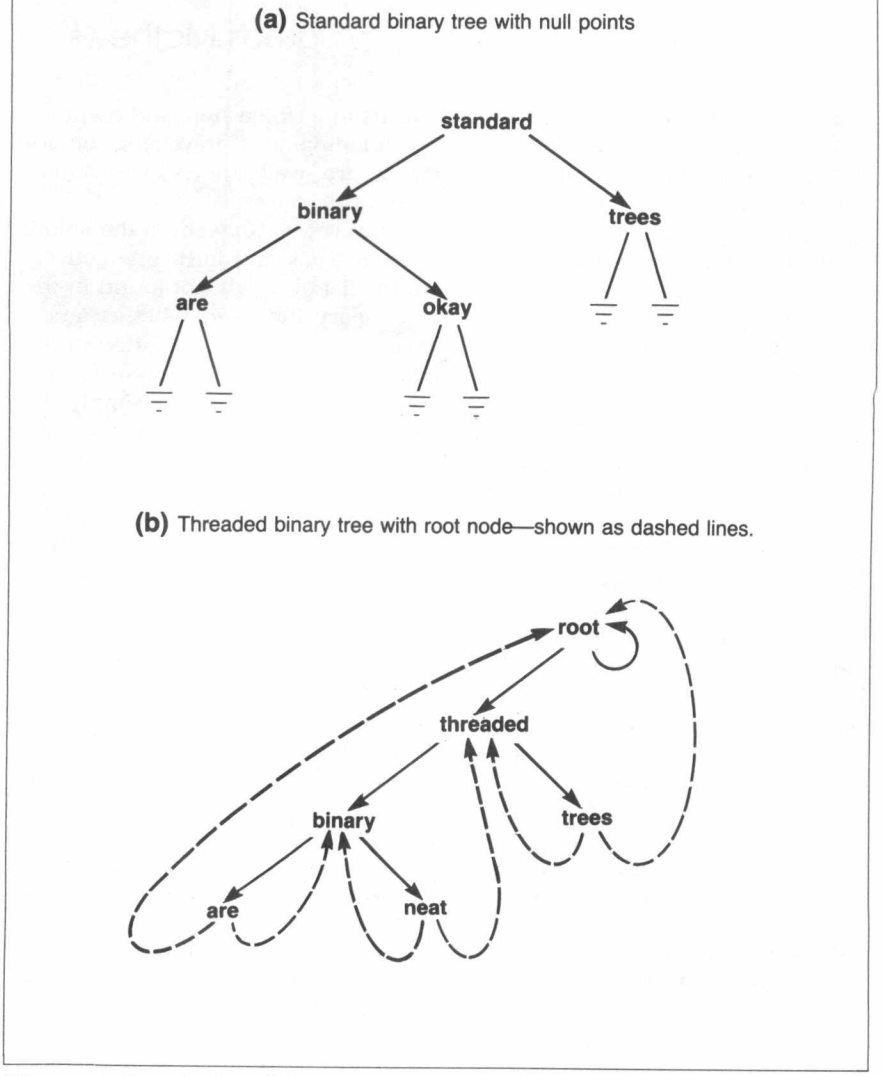


Figure 1: *Standard and threaded binary trees*

(continued from page 43)

subtree of the root node. An empty threaded binary tree consists of just the root node.

The root node contains a data value that is higher than that of any other node in the tree (I chose the ASCII . character as the root node's word value because . is numerically greater than any uppercase or lowercase alphabetic character). This avoids special code in the *add2tree( )* function for dealing with the root node.

The right and left child pointers and the *RBIT* and *LBIT* fields in the root node are initialized such that the root node becomes the in-order predecessor of the lowest-valued tree node and the in-order successor of the highest-valued node. Traversing the tree from start to finish begins and ends at the root node. The definition of the root node occurs in Listing Two just prior to the *add2tree( )* function.

Function *inorder_succ( )* (also in Listing Two) returns the in-order successor of any node in the threaded binary tree. To find the successor node, *inorder_succ( )* looks at the right child of the starting node. If the right child pointer is a thread, then it points to the successor node and *inorder_succ( )* simply returns that node address. If the starting node has an actual right child node, however, the successor is the leftmost child of the starting node's right child. The leftmost child is located by a simple *while* statement that moves down the list of left child pointers.

Function *inorder_pred( )* correspondingly locates the in-order predecessor of any node in the tree. *inorder_pred( )* returns the starting node's left child pointer (if it's a thread) or the rightmost child of the starting node's left child.

To perform an in-order traversal of the entire threaded binary tree, you need only make successive calls to *inorder_succ( )* (for nodes in ascending order by data value) or *inorder_pred( )* (descending order by data value). The following few lines of code, for example, would print out all the words in the tree in ascending order:

```
TREE_NODE *tp;

tp = &root;              /* start at root */
while ((tp = inorder_succ(tp)) !=
                &root) /* end at root */
   puts(tp->word);
```

In Summary

Ease of traversal is the major advantage of a threaded binary tree. There are other well-known algorithms for traversing binary trees, but some of those require the use of recursion or an auxiliary stack to maintain the current location within the tree. Recursive algorithms and those that use an auxiliary stack require additional memory above and beyond the requirements of the tree itself. For applications in which the size and structure of the tree is not known beforehand (such as a spelling checker), trying to ensure that sufficient memory exists for the program or auxiliary stack can be difficult. Because threaded binary trees use the otherwise null pointers as threads, this type of tree can be traversed without additional runtime storage.

There are other algorithms that can traverse a binary tree without recursion or the use of an auxiliary stack, but they typically require the addition of a parent node pointer field to each node or the temporary reversal of the direction of the pointer fields and the addition of a flag to indicate if a particular node has already been processed. The linked list approach of a threaded binary tree is conceptually simpler and easier to implement.

Threaded binary trees are not a new form of data structure. They were documented in 1960 by Perlis and Thornton[3] and are discussed by Knuth[2] in his The Art of Computer Programming series. Threaded binary trees do not seem to have received the widespread recognition or usage that other forms of binary trees have achieved, however. I was prompted to write this article in the hope that I could introduce (or possibly reintroduce) the concepts to others who might find them useful.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Notes

1. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures* (Rockville, Md.: Computer Science Press, 1982).

2. D. Knuth, *The Art of Computer Programming: Fundamental Algorithms, Volume I*, 2d ed (Reading, Mass.: Addison-Wesley, 1973).

3. A. Perlis and C. Thornton, "Symbol Manipulation by Threaded Lists," *Communications of the ACM*, vol. 3, no. 4 (April 1960): 195–204.

DDJ

(Listings begin on page 58.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

ITALIC FONT IN C

Listing One *(Text begins on page 36.)*

```
/* ITALIC.ASC -- This is an ASCII representation of the italic font   */
/* characters used in ITALIC.C.       This file is #includeD.                  */
/* In the table below, each row corresponds to a character.  The      */
/* 14 elements of each row correspond to the 14 scan lines of the     */
/* character.                                                         */
char italic_arr[128-32][14] = {
/* Font character 32 (ASCII value   ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 33 (ASCII value ! ) is */ {0x00, 0x00, 0x06, 0x0f, 0x1e, 0x1e, 0x18, 0x18, 0x00, 0x30, 0x60, 0x00, 0x00, 0x00},
/* Font character 34 (ASCII value " ) is */ {0xc0, 0x0c, 0x99, 0x19, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 35 (ASCII value # ) is */ {0x00, 0x00, 0x1b, 0x1b, 0x7f, 0x36, 0x6c, 0x6c, 0xfc, 0xd9, 0xb0, 0x01, 0x00, 0x00},
/* Font character 36 (ASCII value $ ) is */ {0x03, 0x03, 0x9f, 0xf1, 0x61, 0xe0, 0x7c, 0x06, 0x0c, 0x8d, 0xf0, 0x61, 0xc0, 0x00},
/* Font character 37 (ASCII value % ) is */ {0x00, 0x00, 0x00, 0x00, 0x61, 0xe3, 0x0c, 0x18, 0x60, 0xcc, 0x18, 0x03, 0x00, 0x00},
/* Font character 38 (ASCII value & ) is */ {0x00, 0x00, 0x0e, 0x1b, 0x36, 0x1c, 0x76, 0xdc, 0x98, 0x99, 0xd8, 0x01, 0x00, 0x00},
/* Font character 39 (ASCII value ' ) is */ {0x00, 0x06, 0x0c, 0x0c, 0x30, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 40 (ASCII value ( ) is */ {0x00, 0x00, 0x03, 0x06, 0x18, 0x18, 0x30, 0x30, 0x60, 0x30, 0x30, 0x00, 0x00, 0x00},
/* Font character 41 (ASCII value ) ) is */ {0x00, 0x00, 0x0c, 0x06, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x30, 0xc0, 0x00, 0x00, 0x00},
/* Font character 42 (ASCII value * ) is */ {0x00, 0x00, 0x00, 0x00, 0x33, 0x1e, 0xff, 0x3c, 0xcc, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 43 (ASCII value + ) is */ {0x00, 0x00, 0x00, 0x00, 0x0c, 0x0c, 0x7e, 0x18, 0x30, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 44 (ASCII value , ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x30, 0x60, 0xc0, 0x00, 0x00},
/* Font character 45 (ASCII value - ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xfe, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 46 (ASCII value . ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00},
/* Font character 47 (ASCII value / ) is */ {0x00, 0x00, 0x80, 0x01, 0x06, 0x0c, 0x30, 0x60, 0x80, 0x01, 0x00, 0x00, 0x00, 0x00},
/* Font character 48 (ASCII value 0 ) is */ {0x00, 0x00, 0x1f, 0x71, 0x63, 0xcf, 0xf6, 0xe6, 0x8c, 0x8c, 0xf8, 0x00, 0x00, 0x00},
/* Font character 49 (ASCII value 1 ) is */ {0x00, 0x00, 0x06, 0x0e, 0x3c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0xf8, 0x00, 0x00, 0x00},
/* Font character 50 (ASCII value 2 ) is */ {0x00, 0x00, 0x1f, 0x31, 0x03, 0x06, 0x18, 0x30, 0xc0, 0x8c, 0xf8, 0x00, 0x00, 0x00},
/* Font character 51 (ASCII value 3 ) is */ {0x00, 0x00, 0x1f, 0x11, 0x03, 0x03, 0x3c, 0x06, 0x0c, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 52 (ASCII value 4 ) is */ {0x00, 0x00, 0x03, 0x07, 0x1e, 0x36, 0xcc, 0xfe, 0x18, 0x18, 0x78, 0x00, 0x00, 0x00},
/* Font character 53 (ASCII value 5 ) is */ {0x00, 0x00, 0x3f, 0x30, 0x60, 0x60, 0x7c, 0x06, 0x0c, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 54 (ASCII value 6 ) is */ {0x00, 0x00, 0x0e, 0x18, 0x60, 0x60, 0xfc, 0xc6, 0x8c, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 55 (ASCII value 7 ) is */ {0x00, 0x00, 0x3f, 0x71, 0x03, 0x06, 0x18, 0x30, 0x60, 0xc0, 0xc0, 0x00, 0x00, 0x00},
/* Font character 56 (ASCII value 8 ) is */ {0x00, 0x00, 0x1f, 0x71, 0x63, 0x63, 0x7c, 0xc6, 0x8c, 0xcc, 0xf8, 0x00, 0x00, 0x00},
/* Font character 57 (ASCII value 9 ) is */ {0x00, 0x00, 0x1f, 0x71, 0x63, 0x63, 0x7e, 0x06, 0x0c, 0x18, 0xe0, 0x00, 0x00, 0x00},
/* Font character 58 (ASCII value : ) is */ {0x00, 0x00, 0x00, 0x00, 0x06, 0x0c, 0x00, 0x00, 0x00, 0x30, 0x30, 0x00, 0x00, 0x00},
/* Font character 59 (ASCII value ; ) is */ {0x00, 0x00, 0x00, 0x06, 0x0c, 0x00, 0x00, 0x00, 0x30, 0x30, 0xc0, 0x00, 0x00, 0x00},
/* Font character 60 (ASCII value < ) is */ {0x00, 0x00, 0x01, 0x03, 0x0c, 0x18, 0x60, 0x30, 0x30, 0x18, 0x18, 0x00, 0x00, 0x00},
/* Font character 61 (ASCII value = ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3f, 0x00, 0x00, 0xfc, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 62 (ASCII value > ) is */ {0x00, 0x00, 0x18, 0x0c, 0x0c, 0x06, 0x06, 0x0c, 0x30, 0x60, 0x80, 0x01, 0x00, 0x00},
/* Font character 63 (ASCII value ? ) is */ {0x00, 0x00, 0x9f, 0xf1, 0x63, 0x06, 0x18, 0x18, 0x00, 0x30, 0x60, 0x00, 0x00, 0x00},
/* Font character 64 (ASCII value @ ) is */ {0x00, 0x00, 0x9f, 0xf1, 0x63, 0xef, 0xde, 0xde, 0xb8, 0x81, 0xf0, 0x01, 0x00, 0x00},
/* Font character 65 (ASCII value A ) is */ {0x00, 0x00, 0x04, 0x0e, 0x36, 0x62, 0xc6, 0xfe, 0x8c, 0x8c, 0x98, 0x00, 0x00, 0x00},
/* Font character 66 (ASCII value B ) is */ {0x00, 0x00, 0x3f, 0x1b, 0x33, 0x33, 0x7c, 0x66, 0xcc, 0xcc, 0xf0, 0x00, 0x00, 0x00},
/* Font character 67 (ASCII value C ) is */ {0x00, 0x00, 0x0f, 0x19, 0x61, 0xe0, 0xc0, 0xc0, 0x84, 0xcc, 0xf0, 0x00, 0x00, 0x00},
/* Font character 68 (ASCII value D ) is */ {0x00, 0x00, 0x3e, 0x1b, 0x33, 0x33, 0x66, 0x66, 0xcc, 0xd8, 0xe0, 0x00, 0x00, 0x00},
/* Font character 69 (ASCII value E ) is */ {0x00, 0x00, 0x3f, 0x19, 0x31, 0x30, 0x78, 0x60, 0xc4, 0xcc, 0xfc, 0x00, 0x00, 0x00},
/* Font character 70 (ASCII value F ) is */ {0x00, 0x00, 0x3f, 0x19, 0x31, 0x30, 0x7c, 0x60, 0xc0, 0xc0, 0xc0, 0x00, 0x00, 0x00},
/* Font character 71 (ASCII value G ) is */ {0x00, 0x00, 0x0f, 0x19, 0x61, 0xe0, 0xc0, 0xde, 0x8c, 0xcc, 0xf8, 0x00, 0x00, 0x00},
/* Font character 72 (ASCII value H ) is */ {0x00, 0x00, 0x31, 0x71, 0x63, 0xe3, 0xfe, 0xc6, 0xcc, 0x8c, 0x88, 0x00, 0x00, 0x00},
/* Font character 73 (ASCII value I ) is */ {0x00, 0x00, 0x0f, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0xf0, 0x00, 0x00, 0x00},
/* Font character 74 (ASCII value J ) is */ {0x00, 0x00, 0x07, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x98, 0x98, 0xe0, 0x00, 0x00, 0x00},
/* Font character 75 (ASCII value K ) is */ {0x00, 0x00, 0x39, 0x19, 0x36, 0x36, 0x78, 0x68, 0xc8, 0xc8, 0x98, 0x00, 0x00, 0x00},
/* Font character 76 (ASCII value L ) is */ {0x00, 0x00, 0x3c, 0x18, 0x30, 0x30, 0x60, 0x60, 0xc4, 0xcc, 0xf8, 0x00, 0x00, 0x00},
/* Font character 77 (ASCII value M ) is */ {0x00, 0x00, 0x31, 0x7b, 0x7f, 0xfe, 0xd6, 0xc6, 0x8c, 0x8c, 0x98, 0x00, 0x00, 0x00},
/* Font character 78 (ASCII value N ) is */ {0x00, 0x00, 0x31, 0x79, 0x7b, 0xff, 0xde, 0xce, 0x8c, 0x8c, 0x98, 0x00, 0x00, 0x00},
/* Font character 79 (ASCII value O ) is */ {0x00, 0x00, 0x0e, 0x1b, 0x63, 0xe3, 0xc6, 0xc6, 0x8c, 0xd8, 0xe0, 0x00, 0x00, 0x00},
/* Font character 80 (ASCII value P ) is */ {0x00, 0x00, 0x3f, 0x19, 0x33, 0x33, 0x7c, 0x60, 0xc0, 0xc0, 0xc0, 0x80, 0x00, 0x00},
/* Font character 81 (ASCII value Q ) is */ {0x00, 0x00, 0x1f, 0x71, 0x63, 0xe3, 0xc6, 0xd6, 0xbc, 0xf8, 0x30, 0x38, 0x00, 0x00},
/* Font character 82 (ASCII value R ) is */ {0x00, 0x00, 0x3f, 0x13, 0x33, 0x33, 0x7c, 0x6c, 0xcc, 0xcc, 0x98, 0x00, 0x00, 0x00},
/* Font character 83 (ASCII value S ) is */ {0x00, 0x00, 0x1f, 0x71, 0x63, 0x30, 0x38, 0x0c, 0xcc, 0xcc, 0xf0, 0x00, 0x00, 0x00},
/* Font character 84 (ASCII value T ) is */ {0x00, 0x00, 0x3f, 0x3f, 0x2d, 0x0c, 0x18, 0x18, 0x30, 0x60, 0xf0, 0x00, 0x00, 0x00},
/* Font character 85 (ASCII value U ) is */ {0x00, 0x00, 0x31, 0x71, 0x63, 0xe3, 0xc6, 0xc6, 0xcc, 0xcc, 0xf8, 0x00, 0x00, 0x00},
/* Font character 86 (ASCII value V ) is */ {0x00, 0x00, 0x31, 0x71, 0x63, 0xe3, 0xc6, 0xc6, 0xd8, 0xf0, 0xc0, 0x00, 0x00, 0x00},
/* Font character 87 (ASCII value W ) is */ {0x00, 0x00, 0x31, 0x71, 0x63, 0xc3, 0xd6, 0xd6, 0xdc, 0xf8, 0xb0, 0x00, 0x00, 0x00},
/* Font character 88 (ASCII value X ) is */ {0x00, 0x00, 0x31, 0x71, 0x36, 0x1c, 0x38, 0x78, 0xd8, 0x8c, 0x98, 0x00, 0x00, 0x00},
/* Font character 89 (ASCII value Y ) is */ {0x00, 0x00, 0x1b, 0x1b, 0x32, 0x36, 0x3c, 0x18, 0x30, 0x30, 0xf0, 0x00, 0x00, 0x00},
/* Font character 90 (ASCII value Z ) is */ {0x00, 0x00, 0x3f, 0x71, 0x46, 0x0c, 0x30, 0x60, 0x84, 0x8c, 0xf8, 0x00, 0x00, 0x00},
/* Font character 91 (ASCII value [ ) is */ {0x00, 0x00, 0x0f, 0x0c, 0x18, 0x18, 0x30, 0x30, 0x60, 0x60, 0xf0, 0x00, 0x00, 0x00},
/* Font character 92 (ASCII value \ ) is */ {0x00, 0x00, 0x20, 0xf0, 0x70, 0x38, 0x38, 0x1c, 0x1c, 0x0c, 0x08, 0x00, 0x00, 0x00},
/* Font character 93 (ASCII value ] ) is */ {0x00, 0x00, 0x0f, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0xf0, 0x00, 0x00, 0x00},
/* Font character 94 (ASCII value ^ ) is */ {0x02, 0x07, 0x9b, 0xf1, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 95 (ASCII value _ ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x00},
/* Font character 96 (ASCII value ` ) is */ {0x06, 0x06, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 97 (ASCII value a ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3c, 0x0c, 0x7c, 0x98, 0x98, 0xd8, 0x00, 0x00, 0x00},
/* Font character 98 (ASCII value b ) is */ {0x00, 0x00, 0x38, 0x18, 0x30, 0x3c, 0x6c, 0x66, 0xcc, 0xcc, 0xf0, 0x00, 0x00, 0x00},
/* Font character 99 (ASCII value c ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3e, 0xc6, 0xc0, 0x80, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 100 (ASCII value d ) is */ {0x00, 0x00, 0x07, 0x03, 0x06, 0x1e, 0x6c, 0xcc, 0x98, 0x98, 0xd8, 0x00, 0x00, 0x00},
/* Font character 101 (ASCII value e ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3e, 0xc6, 0xfe, 0x80, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 102 (ASCII value f ) is */ {0x00, 0x00, 0x0e, 0x1b, 0x32, 0x30, 0xf8, 0x60, 0xc0, 0xc0, 0xc0, 0x00, 0x00, 0x00},
/* Font character 103 (ASCII value g ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3b, 0xcc, 0xcc, 0x98, 0xf8, 0x30, 0x30, 0xc0, 0x00},
/* Font character 104 (ASCII value h ) is */ {0x00, 0x00, 0x38, 0x18, 0x30, 0x36, 0x76, 0x66, 0xcc, 0xcc, 0x98, 0x00, 0x00, 0x00},
/* Font character 105 (ASCII value i ) is */ {0x00, 0x00, 0x06, 0x06, 0x00, 0x1c, 0x18, 0x18, 0x30, 0x30, 0xf0, 0x00, 0x00, 0x00},
/* Font character 106 (ASCII value j ) is */ {0x00, 0x00, 0x01, 0x01, 0x00, 0x07, 0x06, 0x06, 0x0c, 0x0c, 0x98, 0x98, 0xe0, 0x00},
/* Font character 107 (ASCII value k ) is */ {0x00, 0x00, 0x38, 0x18, 0x30, 0x33, 0x6c, 0x78, 0xd8, 0xcc, 0x98, 0x00, 0x00, 0x00},
/* Font character 108 (ASCII value l ) is */ {0x00, 0x00, 0x0e, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0xf0, 0x00, 0x00, 0x00},
/* Font character 109 (ASCII value m ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0xf6, 0xfe, 0xd6, 0xac, 0xac, 0x88, 0x00, 0x00, 0x00},
/* Font character 110 (ASCII value n ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0xee, 0x66, 0x66, 0xcc, 0xcc, 0x98, 0x00, 0x00, 0x00},
/* Font character 111 (ASCII value o ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3e, 0xc6, 0xc6, 0x8c, 0x8c, 0xf0, 0x00, 0x00, 0x00},
/* Font character 112 (ASCII value p ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x6e, 0x66, 0x66, 0xcc, 0xf8, 0x80, 0x80, 0x80, 0x00},
/* Font character 113 (ASCII value q ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3b, 0xcc, 0xcc, 0x98, 0xf8, 0x30, 0x30, 0xf0, 0x00},
/* Font character 114 (ASCII value r ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x6e, 0x76, 0x66, 0xc0, 0xc0, 0xc0, 0x00, 0x00, 0x00},
/* Font character 115 (ASCII value s ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x3e, 0x46, 0x70, 0x38, 0x8c, 0xf0, 0x00, 0x00, 0x00},
```

(continued on page 52)

Listing One *(Listing continued, text begins on page 34.)*

```
/* Font character 116 (ASCII value t ) is */ {0x00, 0x00, 0x04, 0x0c, 0x18, 0x7e, 0x30, 0x30, 0x60, 0x6c, 0x70, 0x00, 0x00, 0x00},
/* Font character 117 (ASCII value u ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0xe6, 0xcc, 0xcc, 0x98, 0x98, 0xd8, 0x00, 0x00, 0x00},
/* Font character 118 (ASCII value v ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x33, 0x66, 0x66, 0xcc, 0xf8, 0x60, 0x00, 0x00, 0x00},
/* Font character 119 (ASCII value w ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x63, 0x46, 0xd6, 0xac, 0xfc, 0xb0, 0x00, 0x00, 0x00},
/* Font character 120 (ASCII value x ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x63, 0x6c, 0x38, 0x70, 0xd8, 0x98, 0x00, 0x00, 0x00},
/* Font character 121 (ASCII value y ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x63, 0x66, 0xc6, 0x8c, 0xfc, 0x18, 0x30, 0xe0, 0x00},
/* Font character 122 (ASCII value z ) is */ {0x00, 0x00, 0x00, 0x00, 0x00, 0x7f, 0x4c, 0x18, 0x60, 0xcc, 0xf8, 0x00, 0x00, 0x00},
/* Font character 123 (ASCII value { ) is */ {0x00, 0x00, 0x03, 0x06, 0x0c, 0x0c, 0x70, 0x18, 0x30, 0x30, 0x38, 0x00, 0x00, 0x00},
/* Font character 124 (ASCII value | ) is */ {0x00, 0x00, 0x06, 0x06, 0x0c, 0x0c, 0x00, 0x18, 0x30, 0x30, 0x60, 0x00, 0x00, 0x00},
/* Font character 125 (ASCII value } ) is */ {0x00, 0x00, 0x1c, 0x06, 0x0c, 0x0c, 0x0e, 0x18, 0x30, 0x30, 0xc0, 0x01, 0x00, 0x00},
/* Font character 126 (ASCII value  ) is */ {0x00, 0x00, 0x1d, 0xf7, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
/* Font character 127 (ASCII value  ) is */ {0x00, 0x00, 0x00, 0x00, 0x10, 0x38, 0x6c, 0xc6, 0xc6, 0xfe, 0x00, 0x00, 0x00, 0x00}
} ;  /* End of italic array */
```

End Listing One

Listing Two

```
/* Copyright (C) Magna Carta Software, 1987.  All Rights Reserved.      */
/* MAKEFONT -- Makes an italic font for the EGA.                        */
/* Note: Do not run this program from within the IDE.                   */
/* First version 10/29/87.  Last update: 10/31/87.                      */

#ifndef __SMALL__
    #error Should use SMALL compilation model
#endif

#include <stdio.h>
#include <dos.h>
#include <process.h>
#include <mem.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0


/* Functions related to video operations */
void load_user_egaxfont(char *fptr,int block,int bpc,int char_count,int spos);
void get_egafont(char *fptr, int font);
int get_video_info(void);

/* Global variables and #DEFINES related to video operations */
#include "mk ital.asc"          /* ITALIC.ASC contains our italic font */
#define VIDEO 0x10                              /* BIOS video interrupt */
char fontarray[3585];                           /* buffer for font storage */
int our_id1 = 0xfac, our_id2 = 0x1000;          /* our ID words */
char ega_color;

/* Functions related to TSR operations */
int already_installed(void);
int deinstall(void);
void interrupt (*old_int10h)(void);
void interrupt int10h(unsigned bp, unsigned di, unsigned si,
                      unsigned ds, unsigned es, unsigned dx,
                      unsigned cx, unsigned bx, unsigned ax);

/* Global variables related to TSR operations */
unsigned save_bp1, save_bp2, old_ds, old_psp;
unsigned old_env;


/* Turbo C system variables (see text for explanation ) */
extern unsigned __brklvl;
extern unsigned _psp;


/* Other functions */
void error(int errnum);


main()
{
        int i, j, k;
        union REGS regs;

        get_video_info();
        if (!ega_color) regs.x.ax = 0x7;        /* set the video mode */
        else regs.x.ax = 0x3;
        int86(VIDEO,&regs,&regs);
        if (already_installed()) {
                deinstall();
                printf("\The Italic font is now no longer installed");
                exit(0);
        }

        /* system checks out -- go ahead and put italic chars. in font */
        get_egafont(fontarray,14);      /* store the ROM font in fontarray */
        for(i=14*32,j=0; i< 14*128;j++) {
                for(k=0;k<14;k++) fontarray[i++] = italic_arr[j][k];
        }
        load_user_egaxfont(fontarray,0,14,256,0);       /* load our font */

        old_psp = _psp;                 /* save the resident program's PSP */
        old_env = peek(_psp,0x2c);      /* save the resident program's ENV */
        old_int10h = getvect(VIDEO);    /* save the old vector */
        setvect(VIDEO,int10h);          /* install our INT10h handler */

        /* terminate and stay resident. Program length is determined by */
        /* subtracting the psp address (_psp) from __brkval which is    */
        /* dynamically set to the address of the end of DS.  This       */
        /* appears to be reliable in the TINY and SMALL models, but     */
        /* results are unknown for other models.                        */
        keep(FALSE,_DS + (__brklvl + 15)/16 - _psp);
}


/* ALREADY_INSTALLED: This routine scans through memory for our ID byte.*/
/* Returns: 0 if not found, 1 if found.                                 */
int already_installed()
{
        unsigned int next_seg;

for(next_seg = _psp-1; next_seg > 0; next_seg--) {
        if (peek(next_seg,(unsigned) &our_id1) == our_id1) {
                if (peek(next_seg,(unsigned) &our_id2) == our_id2) {
                        old_ds = next_seg;
                        return (1);
                        }
                }
        }
        return (0);
}


/* DEINSTALL: Remove our TSR by reseting interrupt 0x10 and video mode. */
int deinstall()
{
        union REGS regs;
        struct SREGS sregs;

        /* initialize old interrupt vector */
        old_int10h = MK_FP(peek(old_ds,(unsigned) &old_int10h+2),peek(old_ds,(unsigned)
                                                                &old_int10h));
        setvect(VIDEO,old_int10h);      /* reset the interrupt vector */
        if (!ega_color) regs.x.ax = 0x7;        /* reset the video mode */
        else regs.x.ax = 0x3;
        int86(VIDEO,&regs,&regs);

        /* Deallocate the memory used by the resident program */
        old_psp = peek(old_ds, (unsigned) &old_psp);
        old_env = peek(old_ds, (unsigned) &old_env);

        regs.x.ax = 0x4900;     /* DOS function to free allocated memory */
        sregs.es = old_psp;
        intdosx(&regs,&regs,&sregs);
        if (regs.x.cflag) error(3);

        regs.x.ax = 0x4900;     /* DOS function to free allocated memory */
        sregs.es = old_env;
        intdosx(&regs,&regs,&sregs);
        if (regs.x.cflag) error(4);
        return (0);
}


/* INT10: This function is the BIOS video interrupt handler.            */
void interrupt int10h(unsigned bp, unsigned di, unsigned si,
                      unsigned ds, unsigned es, unsigned dx,
                      unsigned cx, unsigned bx, unsigned ax)

{
        /* execute the old video interrupt */
        if (ax >> 8) {          /* it is not a video mode reset */
                _AX = ax;
                _BX = bx;
                _CX = cx;
                _DX = dx;
                save_bp1 = _BP;
                _BP = bp;
                (*old_int10h)();
                _BP = save_bp1;
                ax = _AX;
                bx = _BX;
                cx = _CX;
                dx = _DX;
        }
        else {                  /* AH == 0 for a video mode reset */
                _AX = ax;
                _BX = bx;
                _CX = cx;
                _DX = dx;
                (*old_int10h)();
                ax = _AX;
                bx = _BX;
                cx = _CX;
                dx = _DX;

                /* reload our font destroyed by the mode reset */
                load_user_egaxfont(fontarray,0,14,256,0);
        }
}


/* LOAD_USER_EGAXFONT -- Load a user-defined font and reset page length.*/
/* Parms: ptr. to user table, block to load, bytes-per-char,            */
/* number of chars to store, starting position in font table.           */
/* First version 7/13/87.  Last update 10/31/87.                        */
void load_user_egaxfont(char *fptr,int block,int bpc,int char_count,int spos)
{
        unsigned byte block:
        byte_block = (bpc << 8) | block;

        /* Can't use intr() due to Turbo C v1.0 compiler bug.           */
        /* Note: we must do any assignments to segments prior to doing  */
        /* assignments to AX since AX is destroyed.                     */
        _ES = _DS;
        _AX = 0x1100;                   /* call function 0x11 */
        _BX = byte_block;               /* block to load */
        _CX = char_count;               /* number of characters to load */
        _DX = spos;                     /* character offset into table */
        save_bp2  = _BP;                /* save BP for stack addressing */
        _BP = FP_OFF(fptr);             /* load address of user font */
        geninterrupt(VIDEO);
        _BP = save_bp2;                 /* restore BP -- or die... */
}
```

```
/* GET_EGAFONT: This routine grabs an EGA font from ROM and stores it   */
/* in the global variable fontarray */
void get_egafont(char *fptr, int font)
{
        struct REGPACK regs;

        regs.r_ax = 0x1130;             /* EGA BIOS call to return font */
        if (font == 8) regs.r_bx = 0x0300;
        else if (font == 14) regs.r_bx = 0x0200;
```

Listing Two *(Listing continued, text begin on page 34.)*

```
        intr(VIDEO,&regs);
        movedata(regs.r_es,regs.r_bp,_DS, (unsigned) fptr,14*256);
}

/* GET_VIDEO_INFO: A VGA or an EGA must be installed for this program  */
/* to work. The monitor must be an Enahanced Color or Monochrome       */
/* display and the correct adaptor must be active.                     */
int get_video_info()
{
        union REGS regs;
        unsigned char e_byte;

        /* First check for the presence of an EGA */
        regs.h.ah = 0x12;       /* EGA BIOS alternate select            */
        regs.h.bl = 0x10;       /* return EGA information.              */
        int86(VIDEO, &regs, &regs);
        if (regs.h.bl == 0x10) error(1);        /* EGA not found */

        /* EGA is present -- is it active? */
        e_byte = peekb(0,0x487);        /* EGA info. byte */
        if (e_byte & 8) error(2);       /* EGA not active */

        /* Does the present, active EGA drive a color or mono monitor? */
        if (regs.h.bh) ega_color = FALSE;  /* EGA drives a mono monitor */
        else ega_color = TRUE;          /* EGA drives a color monitor */

        /* See if EGA drives an Enhanced Color Display */
        if (ega_color) if (!(regs.h.cl == 3 || regs.h.cl == 9)) error(1);
        return (1);
}


/* ERROR: A simple error handler.                                      */
void error(int errnum)
{
        switch (errnum) {
                case 1: printf("\An EGA and Enhanced Color or Monochrome Display");
                        printf("\nmust be present to use this program.");
                        break;

                case 2: printf("\Please make the EGA the active adapter");
                        printf("in order to run this program.");
                        break;

                case 3: printf("\nError deallocating program memory.");
                        break;

                case 4: printf("\nError deallocating program PSP.");
                        break;

                default:break;
        }
        printf("\nProgram exiting.\n");
        exit(0xf);      /* Return code for DOS errorlevel */
}
```

End Listings

BINARY TREES

Listing One *(Text begins on page 42.)*

```
/*  001  23-Apr-87  tbtree.h

    Header file for threaded binary tree functions.

    This code is hereby placed into the public domain.

*/

/* define TREE_NODE type */

typedef struct _tree_ent {
    char *word;                 /* word ptr for this node     */
    int usage;                  /* usage counter for word     */
    int flags;                  /* word flags                 */
    struct _tree_ent *lchild;   /* thread or left child ptr   */
    struct _tree_ent *rchild;   /* thread or right child ptr  */
} TREE_NODE;

/* define bit values for node flags */

#define RBIT (1)            /* right child if set, thread if 0 */
#define LBIT (2)            /* left child if set, thread if 0  */
```

End Listing One

Listing Two

```
/*  001  24-Apr-87  tbtree.c

    Functions to create and traverse a threaded binary tree.

                    James Mathews
                    Blue Sky Software
                    172 Manor Drive
                    Absecon, NJ  08201

    This code is hereby placed into the public domain.

*/
#include <stdio.h>
#include "tbtree.h"

#ifdef LINT_ARGS        /* setup for Microsoft C V 4.0 */
#include <malloc.h>
TREE_NODE *talloc();
void add2tree(char *);
TREE_NODE *linsert(TREE_NODE *);
TREE_NODE *rinsert(TREE_NODE *);
TREE_NODE *inorder_succ(TREE_NODE *);
TREE_NODE *inorder_pred(TREE_NODE *);
#else
char *malloc();
void add2tree();
TREE_NODE *talloc(), *linsert(), *rinsert();
TREE_NODE *inorder_succ(), *indorder_pred();
#endif

/* define the root node for the threaded tree */

TREE_NODE root = { "|", 0, RBIT, &root, &root };


/*********************************************************
                    A D D 2 T R E E
 *********************************************************/

void
add2tree(word)          /* add given word to the B-tree */
char *word;
{
   register int cmp;
   register TREE_NODE *tp = &root;

   /* search tree to find word, add it if not there */

   while ((cmp = stricmp(word,tp->word)) != 0) {

      if (cmp < 0)          /* not equal, look to the left?  */

         if (tp->flags & LBIT)  /* is there a left child?  */
            tp = tp->lchild;    /* yes, go look at it      */
```

Listing Two *(Listing continued, text begins on page 42.)*

```
         else {
            tp = linsert(tp);   /* no left child, make one */
            break;
         }

      else                  /* not left, look right          */

         if (tp->flags & RBIT)  /* is there a right child? */
            tp = tp->rchild;    /* yes, look it over */
         else {
            tp = rinsert(tp);   /* no right child, make one */
            break;
          }
   }

   if (cmp == 0)                /* did we find the word?    */
      tp->usage++;              /*  if so bump usage count  */

   else {                       /* must have added new word */

      tp->word = word;          /* point it to the word     */
      tp->usage = 1;            /* new word in town         */
   }
}


/*********************************************************
                    R I N S E R T
 *********************************************************/

static TREE_NODE *
rinsert(tp)    /* insert new node as right child of tp */
register TREE_NODE *tp;
{
   register TREE_NODE *new;

   if (new = talloc()) {         /* alloc new entry */

      new->rchild = tp->rchild;     /* new gets what tp had */
      new->flags = tp->flags & RBIT;   /* as a right child  */

      new->lchild = tp;          /* lchild is thread to tp */

      tp->rchild = new;          /* new is rchild of tp   */
      tp->flags |= RBIT;         /* real node, not thread */
   }

   return(new);
}


/*********************************************************
                    L I N S E R T
 *********************************************************/

static TREE_NODE *
linsert(tp)    /* insert new node as left child of tp */
register TREE_NODE *tp;
{
   register TREE_NODE *new;

   if (new = talloc()) {         /* alloc new entry */

      new->lchild = tp->lchild;     /* new gets what tp had */
      new->flags = tp->flags & LBIT;     /* as a left child */

      new->rchild = tp;          /* rchild is thread to tp */

      tp->lchild = new;          /* new is lchild of tp   */
      tp->flags |= LBIT;         /* real node, not thread */
   }

   return(new);
}


/*********************************************************
                    T A L L O C
 *********************************************************/

static TREE_NODE *
talloc() {     /* allocate a TREE_NODE */

#define NODES_PER_ALLOC (50)
```

Listing Two *(Listing continued, text begins on page 42.)*

```
    static TREE_NODE *tp;
    static int tidx = NODES_PER_ALLOC;

    /* this routine allocates space for TREE_NODE nodes.  It
       exists to cut down on the number of calls to malloc(). */

    if (tidx < NODES_PER_ALLOC) /* free nodes from last alloc? */
       return(&tp[tidx++]);     /*   yes, return the next one */

    /* allocate another block of TREE_NODE nodes */

    tp = (TREE_NODE *) malloc(sizeof(TREE_NODE) * NODES_PER_ALLOC);

    if (tp == NULL) {
       printf("\nOut of memory in talloc()\n");
       exit();
    }

    tidx = 1;                   /* return # 1 next time */
    return(tp);                 /* return # 0 this time */
}


/*********************************************************
                INORDER_SUCC
 *********************************************************/

TREE_NODE *
inorder_succ(tp)        /* return in-order successor of tp */
register TREE_NODE *tp;
{
   register TREE_NODE *next;

   /* if the right child is a thread, it's the successor node */

   next = tp->rchild;

   /* but if the right child is a real node, the successor is
      left-most child of the right child */

   if (tp->flags & RBIT)                /* real right child? */
      while (next->flags & LBIT)        /*   find left-most  */
         next = next->lchild;           /*   child of that   */

   return(next);                        /* return successor  */
}


/*********************************************************
                INORDER_PRED
 *********************************************************/

TREE_NODE *
inorder_pred(tp)        /* return in-order predecessor of tp */
register TREE_NODE *tp;
{
   register TREE_NODE *prev;

   /* if the left child is a thread, it's the predecessor */

   prev = tp->lchild;

   /* but if the left child is a real node, the predecessor is
      right-most child of the left child */

   if (tp->flags & LBIT)                /* real left child?  */
      while (prev->flags & RBIT)        /*   find right-most */
         prev = prev->rchild;           /*   child of that   */

   return(prev);                        /* return predecessor */
}
```

End Listings

TO THE MACS

Listing One *(Text begins on page 90.)*

```
/* resource decompilation of custom controls PROJ.Rsrc */
/* decompilation performed by Alan Dahlbom's ResTools 2.01 */
/* PICTs, ICONs, ICN#s, and CDEFs not included */

resource  'BNDL' (128)
{
    'CuCo',     0,
    {
    'FREF', {0, 128},
    'ICN#', {0, 128}
    }
};

resource  'CNTL' (1)
{
    {0, 0, 28, 130},
    0,
    visible,
    0,
    0,
    640,
    0x0,
    "Press Me To Quit"
};

resource  'CNTL' (2)
{
    {0, 0, 36, 145},
    6,
    visible,
    18,
    0,
    643,
    0x140000,
    "War Is Peace"
};

resource  'CNTL' (3)
{
    {0, 0, 40, 40},
    0,
    visible,
    0,
    0,
    652,
    0x1e,
    ""
};

resource  'CNTL' (4)
{
    {0, 0, 30, 178},
    2,
    visible,
    14,
    768,
    641,
    0x280000,
    "Evolve Or Dissolve"
};

resource  'CNTL' (5)
{
    {240, 300, 297, 361},
    0,
    visible,
    0,
    0,
    645,
    0x330032.
    "\0x00"
};

resource  'CNTL' (6)
{
    {0, 0, 78, 80},
    0,
    visible,
    0,
    0,
    647,
    0x3d003c,
    ""
};

resource  'CNTL' (7)
{
    {0, 0, 50, 45},
    0,
    visible,
    0,
    0,
    646,
    0x46,
    ""
};

resource  'CNTL' (8)
{
    {0, 0, 200, 116},
    0,
    visible,
    0,
    0,
    644,
    0x50,
    ""
};

resource  'CNTL' (9)
{
    {0, 12, 50, 100},
    0,
    visible,
    0,
    0,
    648,
    0x5a,
    ""
};

resource  'CNTL' (10)
{
    {0, 0, 66, 80},
    0,
    visible,
    0,
    0,
    649,
    0x650064,
    ""
};

resource  'CNTL' (11)
{
    {0, 0, 36, 36},
    0,
    visible,
    0,
    0,
    650,
    0x6e,
    ""
};

resource  'CNTL' (12)
{
    {0, 0, 32, 32},
    0,
    visible,
    0,
    0,
    651,
    0x790078,
    ""
};

resource  'CNTL' (13)
{
    {0, 0, 40, 38},
    0,
    visible,
    0,
    0,
    652,
    0x82,
    "\0x00"
};

resource  'CNTL' (14)
{
    {240, 300, 277, 337},
    0,
    visible,
    0,
    0,
    653,
    0x8d008c,
    "\0x00"
};

resource  'CNTL' (15)
{
    {0, 0, 40, 40},
    0,
    visible,
    0,
    0,
    654,
    0x96,
    "\0x00"
};

resource  'CNTL' (16)
{
    {0, 0, 40, 40},
    0,
    visible,
    0,
    0,
    655,
    0xa100a0,
    "\0x00"
};

resource  'CNTL' (17)
{
    {0, 0, 187, 85},
    0,
    visible,
    0,
    0,
    645,
    0xab00aa,
```

```
    ""
};

resource  'CNTL' (18)
{
    {0, 0, 18, 120},
    3,
    visible,
    10,
    0,
    640,
    0x0,
    "Turn Off Some Buttons"
};

resource  'CNTL' (19)
{
    {0, 0, 18, 150},
    3,
    visible,
    12,
    0,
    642,
    0x0,
    "Turn On Some Buttons"
};

resource  'CNTL' (20)
{
    {0, 0, 24, 24},
    0,
    visible,
    0,
    0,
    645,
    0xc900c8,
    ""
};

resource  'CNTL' (21)
{
    {0, 0, 12, 90},
    3,
    visible,
    9,
    0,
    640,
    0x0,
    "\0xa91987 Stan Krute"
};

userdefined resource  'CuCo' (0)
{
   pstring: "custom controls demo
   version 1.( ©1987 by Stan Krute
   all rights reserved"

};

resource  'DITL' (1)
{
  {
    {238, 326, 266, 456},
      Control {enabled, 1};
    {7, 6, 43, 151},
      Control {enabled, 2};
    {129, 285, 169, 325},
      Control {enabled, 3};
    {174, 197, 204, 375},
      Control {enabled, 4};
    {201, 120, 258, 181},
      Control {enabled, 5};
    {4, 230, 82, 310},
      Control {enabled, 6};
    {12, 169, 62, 214},
      Control {enabled, 7};
    {71, 0, 271, 116},
      Control {enabled, 8};
    {213, 199, 263, 287},
      Control {enabled, 9};
    {69, 141, 135, 221},
      Control {enabled, 10};
    {5, 320, 41, 356},
      Control {enabled, 11};
    {138, 226, 170, 258},
      Control {enabled, 12};
    {90, 238, 130, 276},
      Control {enabled, 13};
    {105, 335, 142, 372},
      Control {enabled, 14};
    {47, 330, 87, 370},
      Control {enabled, 15};
    {148, 142, 188, 182},
      Control {enabled, 16};
    {22, 377, 209, 462},
      Control {enabled, 17};
    {49, 18, 67, 138},
      Control {enabled, 18};
    {214, 305, 232, 455},
      Control {enabled, 19};
    {101, 297, 125, 321},
      Control {enabled, 20};
    {5, 364, 17, 454},
      Control {enabled, 21}
  }
};

resource  'DITL' (210)
{
  {
    {10, 10, 178, 286},
      Picture {enabled, 210}
  }
};

resource  'DLOG' (1)
{
    {0, 0, 272, 462},
    1,
    invisible,
    noGoAway,
    0x0,
    1,
    "New Dialog"
};

resource  'DLOG' (210)
{
    {0, 0, 188, 296},
    1,
    invisible,
    noGoAway,
    0x0,
    210,
    "New Dialog"
};

resource  'FREF' (128)
{
    'APPL',
    0,
    ""
};

resource  'MENU' (1)
{
    1,
    0,
    0xffffffff,
    enabled,
    "custom controls demo",
    {

    }
};

resource  'STR ' (20)
{
    "Peace Is War"
};

resource  'STR ' (40)
{
    "Grow Up Or Blow Up"
};
```

End Listing One

Listing Two

```
asm    rectCDEF.asm Exec   QUED/M 2.04
link   rectCDEF.link       Exec   QUED/M 2.04
RMaker         rectCDEF.R0  Exec   QUED/M 2.04
RMaker         rectCDEF.R1  Exec   QUED/M 2.04
RMaker         rectCDEF.R2  custom controls demo        QUED/M 2.04
```

End Listing Two

Listing Three *(Text begins on page 90.)*

```
*----------------------------- file information -------------------------------*
*                                                                              *
*  rectCDEF.Asm                                                                *
*                                                                              *
*                                                                              *
*  Assembly language source code for several styles of button controls         *
*  The assembled code is meant to be a CDEF resource                           *
*                                                                              *
*  The CDEF provides 16 styles of rectangular button controls                  *
*  A button can: (1)   contain text, an ICON, or a PICT                        *
*                (2)   can be outlined, or shadow-outlined                     *
*                (3)   if it contains an ICON or a PICT, can be bare           *
*                (4)   indicate pressing via inversion or a content change     *
*                                                                              *
*  Edited with QUED/M 2.04                                                     *
*  Compiled under MDS 2.01                                                     *
*                                                                              *
*  Written and ©1987 by Stan Krute. All rights reserved. No part of this       *
*  file, or the object code it leads to, may be reproduced, in any form or     *
*  by any means, without the express written permission of the author and      *
*  copyright holder.                                                           *
*                                                                              *
*  Timestamp:    6:51 pm PST              November 16, 1987                    *
*  Spacestamp:   18617 Camp Creek Road Hornbrook, California   96044          *
*                                                                              *
*  This file looks good in 9 point Courier, QUED/M 2.04 tabs set to 3          *
*                                                                              *
*------------------------------------------------------------------------------*


*------------------------------ using the CDEF --------------------------------*

; details on using the CDEF resource produced by this code

;  a custom control is most easily specified via a CNTL resource,
;     which uses a procID to indicate the CDEF resource ID and variation code

;     for this CDEF, the resource ID is 40

;     there are 16 variations, as follows:

;     variation  content border     highlighting via      proc ID
;     ---------  ------- --------   ----------------      -------
;     0          text    outlined      inversion          640
;     1          text    outlined      content change     641
;     2          text    shadowed      inversion          642
;     3          text    shadowed      content change     643
;     4          PICT    bare          inversion          644
;     5          PICT    bare          content change     645
;     6          PICT    outlined      inversion          646
;     7          PICT    outlined      content change     647
;     8          PICT    shadowed      inversion          648
;     9          PICT    shadowed      content change     649
;     10         ICON    bare          inversion          650
;     11         ICON    bare          content change     651
;     12         ICON    outlined      inversion          652
;     13         ICON    outlined      content change     653
;     14         ICON    shadowed      inversion          654
;     15         ICON    shadowed      content change     655

;  for TEXT variations, select a font via the CNTL's contrlValue field:
;     -1 for the window's font, 0 for the System font, 1 for the application font,
;     anything else a standard Mac font number
;  in the case of a font other than the System or window's font, store the font
;     style in the high byte of the contrlMin field, and the font size in the
;     low byte of the contrlMax field

;  for PICT and ICON variations, store a resource ID indicating the PICT or ICON
;        resource in the low word of the CNTL's refCon field

;  for variations that indicate highlighting via a content change, store a resource
;        ID indicating the highlighted-state STR  (for text variations), PICT, or
;        ICON resource in the high word of the CNTL's refCon field

;  when figuring the size of a text button's boundsRect :
;  be sure to size the button so the text will fit. Successive approximation works
;  well. Rule of thumb for an initial height: 8 greater than the font size.

;  when figuring the size of a PICTure button's CNTL's boundsRect :
;  if the picture has no outline, try a boundsRect that matches the PICT's boundsRect
;  if the picture is outlined, try a boundsRect that's at least 4 wider and 4
;     higher  -- that size lets the picture perfectly match the button's interior
;  pictures in larger boundsRects get centered

;  when figuring the size of a ICON button's CNTL's boundsRect :
;  if the icon has no outline, try a boundsRect that matches the ICON's boundsRect
;  if the icon is outlined, try a boundsRect that's at least 4 wider and 4
;     higher  -- that size lets the icon perfectly match the button's interior
;  icons in larger boundsRects get centered

*------------------------------ include files ---------------------------------*

; standard Mac definitions
   Include MacTraps.D          ; condensed trap file
   Include SysEqu.D            ; system equates
   Include ToolEqu.D           ; toolbox equates
   Include QuickEqu.D          ; toolbox equates

; our stuff
   Include rectCDEFEqu.Txt     ; private definitions for this file


*------------------------------ common entry point ----------------------------*

rectCDEFProc

; provide a stack frame
   LINK     A6,#-autoBytes     ; set frame pointer, with enough
                               ; bytes for automatic variables

; save some registers
   MOVEM.L  D3-D7/A2-A4,-(SP)  ; save some work registers

; the key to 68000 code : fill those registers

   LEA      param(A6),A0       ; A0 points to param
   MOVE.L   (A0)+,D3           ; D3 holds param ( usage varies )
   MOVE.W   (A0)+,D7           ; D7 holds message (operation selector )
   MOVEA.L  (A0)+,A2           ; A2 holds theControl (control record handle )
   CLR.L       D4              ; clear out D4
   MOVE.W   (A0)+,D4           ; varCode into D4
   MOVE.B   varFlagTable(D4),D4; D4 holds a byte of variation flags

; set a default function result of 0, while A0's pointing in the right direction
   CLR.L       (A0)

; lock the control record down
   MOVEA.L A2,A0
   _HLock

; get a pointer to the control record
   MOVE.L   (A2),A2
```

```
; lock down and get a pointer to any control data block
   MOVE.L   contrlData(A2),D0   ; grab the handle
   BEQ      caseOut             ; jump if NIL handle

; we've got a control data block, so lock it
   MOVEA.L D0,A3                ; copy the handle
   MOVEA.L A3,A0                ; lock the block
   _HLock
   MOVEA.L (A3),A3              ; get a pointer to the control data block

caseOut
; case out on the message
; just jump off a table of routine offsets
   ADD.W       D7, D7           ; double the message  integer
   MOVE.W  messageTable(D7),D0; grab a table offset
   JSR      messageTable(D0)    ; jump to messageTable + offset

; clean up and go home
; if there was a control data block, unlock it
   MOVE.L   contrlData(A2),D0   ; grab the handle
   BEQ      unlockRec           ; jump if NIL handle

; we've got a control data block, so unlock it
   MOVEA.L D0,A0                ; move the block's handle into place
   _HUnlock                       and unlock it

unlockRec
; unlock the control record
   MOVEA.L theControl(A6),A0
   _HUnlock

; restore the saved registers
   MOVEM.L  (SP)+,D3-D7/A2-A4

; remove the stack frame
   UNLK     A6

; fetch the return address
   MOVEA.L  (SP)+, A0

; set stack pointer to the function result
   ADD.L       #theResult-param,SP

; we're outta here
   JMP      (A0)


*---------------------------- the message jump table -------------------------*

; there are nine possible messages
messageTable
   DC.W     doDrawCntl-messageTable     ; draw the control
   DC.W     doTestCntl-messageTable     ; test the control
   DC.W     doCalcCCntl-messageTable    ; calculate the control's region
   DC.W     doInitCntl-messageTable     ; do control initialization chores
   DC.W     doDispCntl-messageTable     ; do control disposal chores
   DC.W     doPosCntl-messageTable      ; reposition & update the control
   DC.W     doThumbCntl-messageTable    ; calculate control dragging params
   DC.W     doDragCntl-messageTable     ; drag the control
   DC.W     doAutoTrack-messageTable    ; do the control's action proc


*---------------------------- the variations flag table ----------------------*

; there are sixteen control variations

; eight bits are used to flag eight qualities of a control variation
; from bit 7 (hi) to bit 0 (lo), the bits are symbolically named :
;     textBit - pictBit - iconBit - outBit - shadBit - bareBit - invBit - chngBit

varFlagTable
   DC.B     %10010010                   ; text - outlined - invert
   DC.B     %10010001                   ; text - outlined - content change
   DC.B     %10011010                   ; text - shadowed - invert
   DC.B     %10011001                   ; text - shadowed - content change

   DC.B     %01000110                   ; pict - bare - invert
   DC.B     %01000101                   ; pict - bare - content change
   DC.B     %01010010                   ; pict - outlined - invert
   DC.B     %01010001                   ; pict - outlined - content change
   DC.B     %01011010                   ; pict - shadowed - invert
   DC.B     %01011001                   ; pict - shadowed - content change

   DC.B     %00100110                   ; icon - bare - invert
   DC.B     %00100101                   ; icon - bare - content change
   DC.B     %00110010                   ; icon - outlined - invert
   DC.B     %00110001                   ; icon - outlined - content change
   DC.B     %00111010                   ; icon - shadowed - invert
   DC.B     %00111001                   ; icon - shadowed - content change


*---------------------------------- doDrawCntl -------------------------------*

; the application wants the CDEF to draw the control
doDrawCntl
;  if control is invisible, do nothing
   TST.B       contrlVis(A2)           ; non-zero if the control is visible
   BEQ      drawn                      ; it's invisible, so no need to draw it

; save the entry pen state
   PEA      entryPenState(A6)
   _GetPenState

; normalize the pen state
   _PenNormal

; save a copy of the entry clip region
   CLR.L       -(SP)                   ; get a new region
   _NewRgn
   MOVE.L   (SP),entryClipRgnCopy(A6); copy the entry clip region into it
   _GetClip

shadOutTest
; see if a shadowed outline is to be drawn
   BTST  #shadBit,D4            ; check it out
   BEQ   simpOutTest            ; no shadowed outline, on to the simple outline test
   BSR   doShadOutline          ; draw a shadowed outline
   BSR   shadOutIntClip         ; set a clip rect for the interior
   BRA   interiorDrawing        ; on to the interior tests

simpOutTest
; see if a simple outline is to be drawn
   BTST  #outBit,D4             ; check it out
   BEQ   noOutNoTest            ; if not, on to the no-outline duties
   BSR   doSimpOutline          ; draw a simple outline
   BSR   simpOutIntClip         ; set a clip rect for the interior
   BRA   interiorDrawing        ; on to the interior tests

noOutNoTest
   BSR   noOutIntClip           ; set a clip rect for the interior

interiorDrawing
textTest
; see if the button will contain text
   BTST  #textBit,D4            ; text ?
   BEQ   pictTest               ; no, so next test
   BSR   doTextInterior         ; yes, so draw text interior
   BRA   drawn                  ; and jump on

pictTest
   BTST  #pictBit,D4            ; pict ?
   BEQ   iconNoTest             ; no, so it's an icon button
   BSR   doPictInterior         ; yes, so draw pict interior
   BRA   drawn                  ; and jump on

iconNoTest
   BSR   doIconInterior         ; draw icon interior

drawn
; all drawn
; restore the entry pen state
   PEA      entryPenState(A6)
   _SetPenState

; restore the entry clipping region, and get rid of its holder
   MOVE.L   entryClipRgnCopy(A6),-(SP)
   MOVE.L   (SP),-(SP)
   _SetClip
   _DisposRgn

; so long
   RTS


*---------------------------------- doShadOutline ----------------------------*

; draw a shadowed outline for a button

doShadOutline
; move to the starting point for the horizontal shadow line: (left+2,bottom)
   MOVE.W      contrlRect+left(A2),-(SP)
   ADDQ.W      #0002,(SP)
   MOVE.W      contrlRect+bottom(A2),-(SP)
   _MoveTo

; draw a line to the right side of the horizontal shadow line: (right,bottom)
   MOVE.W      contrlRect+right(A2),-(SP)
   MOVE.W      contrlRect+bottom(A2),-(SP)
   _LineTo

; draw a line to the top of the vertical shadow line: (right,top+2)
   MOVE.W      contrlRect+right(A2),-(SP)
   MOVE.W      contrlRect+top(A2),-(SP)
   ADDQ.W      #$0002,(SP)
   _LineTo

; now draw an outline rect at (top,left,bottom,right)
   PEA         contrlRect(A2)
   _FrameRect

; all done
   RTS

*---------------------------------- doSimpOutline ----------------------------*

; draw a simple outline for a button

doSimpOutline

   draw an outline rect at (top,left,bottom,right)
   PEA         contrlRect(A2)
   _FrameRect

; all done
   RTS

*---------------------------------- shadOutIntClip ---------------------------*
; set the clip region for the interior of a shadowed outlined button

shadOutIntClip

  use the rect at (top+2,left+2,bottom-2,right-2)
   MOVE.L      contrlRect+top(A2),interiorClipRect+top(A6)
   MOVE.L      contrlRect+bottom(A2),interiorClipRect+bottom(A6)
   ADDI.L      #$00020002,interiorClipRect+top(A6)
   SUBI.L      #$00020002,interiorClipRect+bottom(A6)
   PEA         interiorClipRect(A6)
   _ClipRect

  all done
   RTS


*---------------------------------- simpOutIntClip ---------------------------*

; set the clip region for the interior of a simply outlined button

simpOutIntClip

; use the rect at (top+2,left+2,bottom-2,right-2)
   MOVE.L      contrlRect+top(A2),interiorClipRect+top(A6)
   MOVE.L      contrlRect+bottom(A2),interiorClipRect+bottom(A6)
   ADDI.L      #$00020002,interiorClipRect+top(A6)
   SUBI.L      #$00020002,interiorClipRect+bottom(A6)
   PEA         interiorClipRect(A6)
   _ClipRect

; all done
   RTS


*---------------------------------- noOutIntClip -----------------------------*

; set the clip region for the interior of a non-outlined button

noOutIntClip
```

```
; use the rect at (top,left,bottom,right)
   MOVE.L     contrlRect+top(A2),interiorClipRect+top(A6)
   MOVE.L     contrlRect+bottom(A2),interiorClipRect+bottom(A6)
   PEA        interiorClipRect(A6)
   _ClipRect

; done
   RTS

*-------------------------------- doTextInterior ----------------------------*

; draw a button's text content, in an indicated font

doTextInterior

; get a pointer to the current grafPort into A4
   PEA        currentGrafPort(A6)
   _GetPort
   MOVEA.L    currentGrafPort(A6),A4

;  we may be using a font other than the window's, so save current font settings
   MOVE.L     txFont(A4),curFontAndFace(A6)  ; font and style
   MOVE.W     txSize(A4),curSize(A6)         ; size

; the control's font is indicated by a value in the control record's contrlValue
;  field:  -1 for the window's font, 0 for the System font, 1 for the application
;  font, anything else a standard Mac font number

; in the case of a font other than the System or window's font, the font style
;  is in the ContrlMin field, and the font size is in the contrlMax field

; case out on the font
   MOVE.W     contrlValue(A2),D0
   BMI        useWindowsFont

; see if the new font is the System font or something else
   TST.W         D0
   BEQ        useSystemFont

useCustomFont
; set the font, style, and size for a font other than the System or window's font
   MOVE.W     D0,txFont(A4)                  ; set the font
   MOVE.W     contrlMin(A2),txFace(A4)       ; set the style
   MOVE.W     contrlMax(A2),txSize(A4)       ; set the size
   BRA        figgerFontInfo

useSystemFont
;  set the font, style, and size for the system font (all zeroes will do it)
   CLR.L         txFont(A4) ; set the font and style
   CLR.W         txSize(A4) ; set the size

useWindowsFont
; we're using the window's current font, so font number, size, and style already set

figgerFontInfo
; so the font's set up -- let's get some more information
   PEA        fontInfo(A6)
   _GetFontInfo

; from that info, we can figure the vertical positioning for the button's text
; the equation: vertPos = rectBottom - (fontDescent + ((rectHeight-fontHeight)/2))
   MOVE.W  interiorClipRect+bottom(A6),D0    ; bottom - top gives rectHeight
   MOVE.L  D0,D7                             ; bottom will be used again
   SUB.W      interiorClipRect+top(A6),D0
   SUB.W      fontInfo+ascent(A6),D0         ; then subtract fontHeight
   SUB.W      fontInfo+descent(A6),D0
   ASR.W      #1,D0                          ; divide what's left by 2
   ADD.W      fontInfo+descent(A6),D0        ; add it to the descent
   SUB.W      D0,D7                          ; then subtract it from bottom

; determine whether we'll be drawing the control's title or a STR resource
; if the control hilites via a content change and is hilited and there's a handle to
;  the STR resource, we draw the STR resource
; content change ?
   BTST     #chngBit,D4
   BEQ      useTitle              ; no content change, so use control's title
; control hilites via a content change
; is it hilited ?
   MOVE.B   contrlHilite(A2),D0
   BEQ      useTitle              ; 0=active, not hilited, so use control's title
   ADDQ.B   #1,D0
   BEQ      useTitle              ; 255=inactive, not hilited, so use control's title

; control hiltes via a content change, and it's hilited
; do we have a handle to the content change string ?
   MOVE.L   firstRsrcHndl(A3),D5
   BEQ      useTitle              ; no, so use control's title

useSTR
; okay, we'll be using a content change STR resource, and we have a non-NIL handle
; lock the resource, put a pointer to it into D5, and set a flag
   MOVEA.L D5,A0                  ; handle into A0
   _HLock                         ; lock the STR resource
   MOVEA.L D5,A0
   MOVE.L   (A0),D5               ; set a pointer to it
   MOVE.W   #1,usingCCRsrc(A6)    ; set a flag
   BRA      setStrWidth

useTitle
; using the control's title
; put a pointer to the string into D5, and clear a flag
   LEA      contrlTitle(A2),A0
   MOVE.L   A0,D5                 ; set a pointer
   CLR.W    usingCCRsrc(A6)       ; clear a flag

setStrWidth
; now, the horizontal positioning: start by getting the button's string's width
   SUBQ.L   #2,SP                 ; room for function result
   MOVE.L   D5,-(SP)              ; the string
   _StringWidth

; now, use that width to get the horizontal positioning
; the equation: horzPos = rectLeft +  ( (rectWidth-stringWidth) / 2)
   MOVE.W  interiorClipRect+right(A6),D0     ; get rect width  ( right - left )
   SUB.W      interiorClipRect+left(A6),D0
   SUB.W      (SP)+,D0                       ; subtract string width
   ASR.W      #1,D0                          ; divide what's left by two
   ADD.W      interiorClipRect+left(A6),D0   ; and add in the left side of rect

; now we have the horizontal and vertical starting position for string drawing
; let's move there ...
   MOVE.W     D0,-(SP)            ; the horizontal starting position
   MOVE.W     D7,-(SP)            ; the vertical starting position
   _MoveTo

; paint a button's interior's background, according to the button's hilite state
;  and whether we're drawing a content change

; test for content change imminent
   TST.W       usingCCRsrc(A6)     ; remember, we just set or cleared this flag above

   BNE      cCImminent1            ; flag set, content change imminent

; test the hilite state
bkgHSTest1
   MOVE.B   contrlHilite(A2),D7
   BEQ      hSActive1             ; 0 indicates an active button

bkgHSTest2
   ADDQ.B   #1,D7                 ; 255 indicates an inactive button
   BNE      hSHilit1              ; 1-253 indicates a highlighted button

hSInactive1
; the button is inactive, so paint interior background white

cCImminent1
; a content change is imminent, so paint interior background white

hSActive1
; the button is active, so paint interior background white
   MOVEA.L    (A5),A0
   PEA        white(A0)
   _PenPat

hSHilit1
; the button is highlighted, so paint interior background black

paintInteriorBkg
; paint the button's interior background
   PEA      interiorClipRect(A6)
   _PaintRect

; draw the button's interior's text, according to the button's hilite state
; and whether we're drawing a content change
```

Listing Three *(Listing continued, text begins on page 90.)*

```
; clear a flag that, if set, signals an inactive button
   CLR.B      D6

; test for content change imminent
   TST.W      usingCCRsrc(A6)
   BNE        cCImminent2          ; flag set, content change imminent

; test the hilite state
txtIntHSTest1
   MOVE.B   contrlHilite(A2),D7
   BEQ      txtHSActive2          ; 0 indicates an active button

txtIntHSTest2
   ADDQ.B   #1,D7                 ; 255 indicates an inactive button
   BNE      txtHSHilit2           ; 1-253 indicates a highlighted button

txtHSInactive2
; the button is inactive, so we'll draw the text in  black,
                                          then gray it out
   ADDQ.B   #1,D6                 ; set the inactive flag
   BRA      drawText              ; and jump to draw

txtHSHilit2
; the button is highlighted, so we'll draw the text in white

   MOVE.W   #srcBic,txMode(A4)  ; so the black bits show as white

cCImminent2
; a content change is imminent, so we'll draw the text in black

txtHSActive2
; the button is active, so we'll draw the text in black

drawText
; draw the button's string
   MOVE.L     D5,-(SP)           ; pointer to the string to draw
   _DrawString

; if we used a STR resource, unlock it
   TST.W      usingCCRsrc(A6)  ; are we using ?
   BNE      grayTest                ; no, so jump ahead
   MOVE.L  firstRsrcHndl(A3),A0     ; yes, so get the handle
   _HUnlock                         ; and unlock it

grayTest
; see if we've got an inactive button, in which case we gray the text
                                                                    out
   TST.B      D6
   BEQ      txtGetNorm          ; not inactive, so no graying out

; yup, we're inactive, so let's get grayed out
; set the pen pattern to gray
   MOVEA.L     (A5),A0
   PEA         gray(A0)
   _PenPat

; and set pattern transfer mode to ANDing with the inverse of the
                                                              pattern
   MOVE #PatBic,-(SP)
   _PenMode

; now paint the clip rectangle gray
   PEA      interiorClipRect(A6)
   _PaintRect

txtGetNorm
; get things back to normal, in case they were changed
; normalize the text transfer mode
   MOVE.W      #srcOr,txMode(A4)

;restore the window's prior font settings
   MOVE.L      curFontAndFace(A6),txFont(A4)
   MOVE.W      curSize(A6),txSize(A4)

textDrawn
; text button interior's all drawn
   RTS


*------------------------ doPictInterior ---------------------------*

; draw a PICTure interior for the button

doPictInterior
; we'll draw in the picture's bounding rectangle
; we'll try to center this rectangle horizontally and vertically
;  in the control's clipping rectangle
; if the control's too small, the icon lines up against
;  the top and or left sides of the control's clipping rectangle

; move clipping rectangle's top and left coords into place
   MOVE.L   interiorClipRect+top(A6),pictRect+top(A6)

; get the clipping rect's width into D6
   MOVE.W   interiorClipRect+right(A6),D6
   SUB.W      interiorClipRect+left(A6),D6

; get a pointer and handle to the picture
; if the control hilites via a content change and is hilited,
                                          we draw the secondary
;  PICT  resource
; does the control hilite via a content change ?
   BTST     #chngBit,D4
   BEQ      usePictOne             ; doesn't hilite via a content
                                                          change
; is the control hilited ?
   MOVE.B   contrlHilite(A2),D1
   BEQ      usePictOne             ; 0=active, not hilited
   ADDQ.B   #1,D1
   BEQ      usePictOne             ; 255=inactive, not hilited

usePictTwo
; we'll be using the secondary PICT resource
; get a handle and set a flag
   MOVE.L   secondRsrcHndl(A3),D5 ; the handle
   BEQ      usePictOne             ; in case of a NIL handle
   MOVE.W   #1,usingCCRsrc(A6)     ; the flag
   BRA      getPictPointer

usePictOne
; we'll be using the primary PICT resource
; get a handle and set a flag
   MOVE.L   firstRsrcHndl(A3),D5  ; the handle
   BEQ      pictDrawn              ; in case of a NIL handle
   CLR.W      usingCCRsrc(A6)     ; the flag

getPictPointer
; lock the PICT, and get a pointer to it
   MOVEA.L  D5,A0
   _HLock                          ; lock the PICT
   MOVEA.L  D5,A4
   MOVEA.L  (A4),A4                ; get a pointer

figPicWidth
; figure the picture's width
   MOVE.W   picFrame+right(A4),D1
   SUB.W      picFrame+left(A4),D1

; now subtract the picture's width from the clip width
   SUB.W      D1,D6

; if 2 or more, divide by 2 and use as horizontal offset

; if it's < 2,  no horizontal offset
   CMPI.W   #2,D6
   BLT.S      doPictRight

; divide and add the offset in
   ASR.W      #1,D6               ; divide by two
   ADD.W      D6,pictRect+left(A6)

; now do the right coord of rectangle
doPictRight
   MOVE.W   pictRect+left(A6),pictRect+right(A6)
   ADD.W      D1,pictRect+right(A6)

; get the clipping rect's height
   MOVE.W   interiorClipRect+bottom(A6),D0    ; get clip height into D0
   SUB.W      interiorClipRect+top(A6),D0

; figure the picture's height
   MOVE.W   picFrame+bottom(A4),D1
   SUB.W      picFrame+top(A4),D1

; now subtract the picture's height from the clip height
   SUB.W      D1,D0

; if 2 or more, divide by 2 and use as vertical offset
; if it's < 2,  no vertical offset
   CMPI.W   #2,D0
   BLT.S      doPictBottom

; divide and add the offset in
   ASR.W      #1,D0                          ; divide by two
   ADD.W      D0,pictRect+top(A6)

; now do the bottom coord of rectangle
doPictBottom
   MOVE.W   pictRect+top(A6),pictRect+bottom(A6)
   ADD.W      D1,pictRect+bottom(A6)

; clear the background
   PEA      interiorClipRect(A6)
   _EraseRect
```

```
; okay, let's draw  the picture (remember, it gets clipped to the button's interior )
   MOVE.L     D5,-(SP)          ; the picture's handle
   PEA        pictRect(A6)      ; the destination rectangle
   _DrawPicture

; unlock the PICT
   MOVEA.L  D5,A0               ; handle's still here
   _HUnlock

; now, adjust the picture for buttons that are hilited inverts or inactive
   BSR      interiorAdjust

pictDrawn
; pict button interior's all drawn
   RTS

*---------------------------------- interiorAdjust----------------------------*

; adjust a button's interior for buttons that are hilited inverts or inactive

interiorAdjust

; test the hilite state
hiliteTest1
   MOVE.B   contrlHilite(A2),D0
   BEQ      intAdjDone     ; 0 indicates a non-hilited active button

hiliteTest2
   ADDQ.B   #1,D0          ; 255 indicates an inactive button
   BNE      itsHilited     ; 1-253 indicates a highlighted button

itsInactive
; the button is inactive, so we'll gray it out
; set the pen pattern to gray
   MOVEA.L     (A5),A0
   PEA         gray(A0)
   _PenPat

; and set pattern transfer mode to ANDing with the inverse of the pattern
   MOVE  #PatBic,-(SP)
   _PenMode

; now paint the clip rectangle gray
   PEA      interiorClipRect(A6)
   _PaintRect

;  leave
   BRA   intAdjDone

itsHilited
; the button is highlighted
; test for being an invert
   TST.W       usingCCRsrc(A6)
   BNE      intAdjDone       ; d content changer, so we done

; the button's a hilited invert, so invert it
   PEA      interiorClipRect(A6)
   _InverRect

intAdjDone
; all done with the adjustment
   RTS


*---------------------------------- doIconInterior ----------------------------*

; draw an ICONic interior for the button

doIconInterior

; we'll draw in an icon-sized rectangle
; we'll try to center this rectangle horizontally and vertically
;  in the control's clipping rectangle
; if the control's too small, the icon lines up against
;  the top and or left sides of the control's clipping rectangle

; move clipping rectangle's top and left coords into place
   MOVE.L   interiorClipRect+top(A6),iconRect+top(A6)

; get the clipping rect's width into D0
   MOVE.W   interiorClipRect+right(A6),D0
   SUB.W       interiorClipRect+left(A6),D0

; now subtract the icon width
   SUBI.W   #iconSize,D0

; if 2 or more, divide by 2 and use as horizontal offset
; if it's < 2,  no horizontal offset
   CMPI.W   #2,D0
   BLT.S       doIconRight
; divide and add the offset in
   ASR.W       #1,D0                        ; divide by two
   ADD.W       D0,iconRect+left(A6)

; now do the right coord of rectangle
doIconRight
   MOVE.W   iconRect+left(A6),iconRect+right(A6)
   ADD.W       #iconSize,iconRect+right(A6)

; get the clipping rect's height
   MOVE.W   interiorClipRect+bottom(A6),D0    ; get clip height into D0
   SUB.W       interiorClipRect+top(A6),D0

; now subtract the icon height from the clip height
   SUBI.W      #iconSize,D0

; if 2 or more, divide by 2 and use as vertical offset
; if it's < 2,  no vertical offset
   CMPI.W   #2,D0
   BLT.S       doIconBottom

; divide and add the offset in
   ASR.W       #1,D0                        ; divide by two
   ADD.W       D0,iconRect+top(A6)

; now do the bottom coord of rectangle
doIconBottom
   MOVE.W   iconRect+top(A6),iconRect+bottom(A6)
   ADD.W       #iconSize,iconRect+bottom(A6)

; get a handle to the icon
; if the control hilites via a content change and is hilited, we draw the secondary
;  ICON  resource
; content change ?
   BTST     #chngBit,D4
   BEQ      useIconOne
; hilited ?
   MOVE.B   contrlHilite(A2),D1
   BEQ      useIconOne
   ADDQ.B   #1,D1
   BEQ      useIconOne

useIconTwo
; we'll be using the secondary ICON resource
; get a handle and set a flag
   MOVE.L   secondRsrcHndl(A3),D5 ; the handle
   BEQ      useIconOne            ; in case of a NIL handle
   MOVE.W   #1,usingCCRsrc(A6)    ; the flag
   BRA      clearBack             ; and jump ahead

useIconOne
; we'll be using the primary ICON resource
; get a handle and set a flag
   MOVE.L   firstRsrcHndl(A3),D5  ; the handle
   BEQ      iconDrawn             ; in case of a NIL handle
   CLR.W       usingCCRsrc(A6)    ; the flag

clearBack
; clear the background
   PEA      interiorClipRect(A6)
   _EraseRect

; okay, let's draw  the icon  (remember, it gets clipped to the button's interior )
   PEA         iconRect(A6)       ; the destination rectangle
   MOVE.L      D5,-(SP)           ; the icon's handle
   _PlotIcon

; now, adjust the icon for buttons that are hilited inverts or inactive
   BSR      interiorAdjust

iconDrawn
; icon button interior's all drawn
   RTS


*---------------------------------- doTestCntl ----------------------------*

;  the application wants CDEF to test a point to see if it's in an active control

doTestCntl

; first, see if the control's even active, or if it's in one of the two inactive modes
   MOVE.B      contrlHilite(A2),D7   ;inactive control ?
   CMPI.B      #inact254,D7
   BEQ         setReturn2          ;yes, so return appropriate code
   CMPI.B      #inact255,D7
   BEQ         setReturn3          ;yes, so return appropriate code

; the control has been found to be active
; now find out if the supplied point is in the control's rectangle
   SUBQ.L      #2,SP               ; room for the function result
   MOVE.L      D3,-(SP)            ; param holds the mouse coords
   PEA         contrlRect(A2)      ; pointer to the control's rectangle
   _PtInRect
   TST.B          (SP)+            ; scan result while chucking
   BEQ         setReturn3          ; if point isn't in the control ...

; okay, the control's active, and the mouse point's in it
; the control has no separate parts,  so we return the whole control's part number
setReturn1
   MOVE.L      #wholePartNumber,theResult(A6)
   RTS

; and here are the return codes for the other possibilities
setReturn2
   MOVE.L      #254,theResult(A6)  ;inactive type 254
   RTS

setReturn3
   MOVE.L      #0,theResult(A6)    ;inactive type 255
   RTS                             ;or not in  control


*---------------------------------- doCalcCCntl ----------------------------*

;  calculate the control's region


doCalcCCntl
   MOVE.L      D3,-(SP)            ; param holds the waiting handle
   PEA         contrlRect(A2)      ; use the control's rectangle
   _RectRgn
   RTS


*---------------------------------- doInitCntl ----------------------------*

; do any special initialization of the control

; in this case, set up a control data block
; then load in any necessary resources, and store handles in the control data block

doInitCntl

; try to get a control data block
   MOVE.L   #cntlDataBlokSize,D0
   _NewHandle,CLEAR

; store the result
   MOVE.L   A0,contrlData(A2)
   BEQ      initDone            ; if we got no block, leave

lockDataBlock
   MOVEA.L A0,A3               ; save a copy of the block's handle
   _Hlock                      ; lock the block down
   MOVEA.L (A3),A3             ; get a pointer to the locked block

initTextTest
; is this is a text control ?
   BTST     #textBit,D4         ; the test
   BEQ      initPictTest        ; no, so jump on

;got a text control - does it indicate hiliting via a content change ?
   BTST     #chngBit,D4         ; the test
   BEQ      initPictTest  ; no, so jump on
```

```
; got a text control with content change hiliting, so load in the string resource and lock
it
   SUBQ.L   #4,SP                    ; room for a handle
   MOVE.L   #'STR ',-(SP)            ; the resource type
   MOVE.W      contrlRFcon(A2),-(SP) ; the resource id
   _GetResource                      ; try to grab that resource
   MOVE.L   (SP)+,firstRsrcHndl(A3)  ; store its handle (possibly NIL)
   BRA      initDone

initPictTest
; see if this is a PICT control
   BTST     #pictBit,D4              ; the test
   BEQ      initIconTest             ; no, so jump on

; got a pict control, so load in the main PICT
   SUBQ.L   #4,SP                    ; room for a handle
   MOVE.L   #'PICT',-(SP)            ; the resource type
   MOVE.W      contrlRFcon+2(A2),-(SP)  ; the resource id
   _GetResource                      ; try to grab that resource
   MOVE.L   (SP)+,firstRsrcHndl(A3)  ; store its handle (possibly NIL)

isPictCC
; does it indicate hiliting via a content change ?
   BTST     #chngBit,D4              ; the test
   BEQ      initDone                 ; no, so done
; got a pict control with content change, so load in the secondary PICT
   SUBQ.L   #4,SP                    ; room for a handle
   MOVE.L   #'PICT',-(SP)            ; the resource type
   MOVE.W      contrlRFcon(A2),-(SP)  ; try to grab that resource
   MOVE.L   (SP)+,secondRsrcHndl(A3) ; store its handle (possibly NIL)
   BRA      initDone                 ; done

initIconTest
; no test needed; we have an icon control, so load in the main ICON
   SUBQ.L   #4,SP                    ; room for a handle
   MOVE.L   #'ICON',-(SP)            ; the resource type
   MOVE.W      contrlRFcon+2(A2),-(SP)  ; the resource id
   _GetResource                      ; try to grab that resource
   MOVE.L   (SP)+,firstRsrcHndl(A3)  ; store its handle (possibly NIL)

isIconCC
; does it indicate hiliting via a content change ?
   BTST     #chngBit,D4              ; the test
   BEQ      initDone                 ; no, so done

; got an icon control with content change, so load in the secondary ICON
   SUBQ.L   #4,SP                    ; room for a handle
   MOVE.L   #'ICON',-(SP)            ; the resource type
   MOVE.W      contrlRFcon(A2),-(SP)  ; the resource id
   _GetResource                      ; try to grab that resource
   MOVE.L   (SP)+,secondRsrcHndl(A3) ; store its handle (possibly NIL)

initDone
; that's it for initialization
   RTS

*-------------------------------------- doDispCntl ----------------------------0.*

; do any special disposal operations for the control

; in this case, release resources whose handles are stored in the control's
;  data block, then release that block

doDispCntl
; see if we ever got a control data block
   TST.L       contrlData(A2)
   BEQ      dispDone                 ; no block, so leave

checkFirst
; see if there's a handle in the first slot
   TST.L       firstRsrcHndl(A3)     ; got a real handle ?
   BEQ      checkSecond              ; no, it's NIL, so jump ahead
   MOVE.L      firstRsrcHndl(A3),-(SP)  ; handle okay, so let go of that resource
   _ReleaseResource

checkSecond
; see if there's a handle in the second slot
   TST.L       secondRsrcHndl(A3)    ; got a real handle ?
   BEQ      dropDataBlock            ; no, it's NIL, so jump ahead
   MOVE.L      secondRsrcHndl(A3),-(SP) ; handle okay, so let go of that resource
   _ReleaseResource

dropDataBlock
; now get rid of the control's data block
   MOVEA.L contrlData(A2),A0
   _DisposHandle

dispDone    RTS

*-------------------------------------- doPosCntl -----------------------------*

;  the position routine
;  in this case, do nothing

doPosCntl      RTS


*-------------------------------------- doThumbCntl ---------------------------*

;  the thumb routine
;  in this case, do nothing

doThumbCntl    RTS


*-------------------------------------- doDragCntl ----------------------------*

;  the drag routine
;  in this case, do nothing

doDragCntl     RTS


*-------------------------------------- doAutoTrack ---------------------------*

;  the track routine
;  in this case, do nothing

doAutoTrack    RTS
```

End Listing Three

Listing Four

```
*---------------------------------- file information ------------------------*
*                                                                            *
*  rectCDEFEqu.Txt                                                           *
*                                                                            *
*                                                                            *
*  Private definitions for rectCDEF.Asm                                      *
*                                                                            *
*  Edited with QUED/M 2.04                                                   *
*  Compiled under MDS 2.01                                                   *
*                                                                            *
*  Written and ©1987 by Stan Krute. All rights reserved. No part of this file,*
*  or the object code it leads to, may be reproduced, in any form or by any  *
*  means, without the express written permission of the author and copyright  *
*  holder.                                                                   *
*                                                                            *
*  Timestamp:    1:56 am EST          September 29, 1987                     *
*  Spacestamp:   21E Halcyon Drive  West Yarmouth, Massacusetts    02673     *
*                                                                            *
*  This file looks good in 9 point Courier, QUED/M 2.04 tabs set to 3        *
*                                                                            *
*----------------------------------------------------------------------------*

*---------------------------------- equates ---------------------------------*

; stack frame offsets for function parameters
returnAddress    EQU     4       ; return address' offset in frame
param            EQU     8       ; for long-word-size parameter
message          EQU     12      ; control message identifies desired operation
theControl       EQU     14      ; calling control's handle's offset in frame
varCode          EQU     18      ; which variation of the control
theResult        EQU     20      ; function result offset in frame

; stack frame offsets for automatic (local) variables
entryPenState    EQU     -18     ;  room to hold entry pen state (18 bytes)
currentGrafPort  EQU     -22     ; pointer to current grafPort ( 4 bytes )
curFontAndFace   EQU     -26     ; saved font number and style ( 4 bytes )
curSize          EQU     -28     ; saved font size ( 2 bytes )
fontInfo         EQU     -36     ; information about current font ( 8 bytes )
entryClipRgnCopy EQU     -40     ; handle to copy of entry clip region (4 bytes)
interiorClipRect EQU     -48     ; a clipping rectangle (8 bytes)
pictRect         EQU     -56     ; a PICTure bounding rectangle (8 bytes)
iconRect         EQU     -56     ; an ICON bounding rectangle (8 bytes)
usingCCRsrc      EQU     -58     ; flags use ofcontent change resource (2 bytes)
autoBytes        EQU     58      ; size in bytes of automatic variable area

; hilite codes
inact254         EQU     254     ; hilite code to inactivate control
inact255         EQU     255     ; hilite code to inactivate control

; icon stuff
iconSize         EQU     32      ; width and height of icon

; id's for our control definition
wholePartNumber  EQU     10      ; part number for our whole control

; test bits
textBit          EQU     7       ; if set, it's a text button
pictBit          EQU     6       ; if set, it's a PICT button
iconBit          EQU     5       ; if set, it's an ICON button
outBit           EQU     4       ; if set, the button is outlined
shadBit          EQU     3       ; if set, the button's outline is shadowed
bareBit          EQU     2       ; if set, the button has no outline
invBit           EQU     1       ; if set, the button shows hiliting via inversion
chngBit          EQU     0       ; if set, the button shows hiliting via content change

; the control's data block
cntlDataBlokSize EQU     8       ; size of the control's data block
firstRsrcHndl    EQU     0       ; offset of first data block field
secondRsrcHndl   EQU     4       ; offset of second data block field
```

End Listing Four

Listing Five

```
;    this file is called rectCDEF.Link

; ©1987 by Stan Krute   --   all rights reserved

; timestamp:    12:57 am EST       September 26, 1987
; spacestamp:   21E Halcyon Drive  West Yarmouth, Mass.     02673

; turn off code listing to map file
]
;  the name of the input file is rectCDEF.Rel
rectCDEF.REL

;  the name of the output file is rectCDEF
/Output  rectCDEF

; set a type and creator for the output file
/Type    '????'  '????'

;  end of Linker control file
$
```

End Listing Five

Listing Six

```
*    this file is called rectCDEF.R0

*    ©1987 by Stan Krute   --   all rights reserved

*    timestamp:     12:53 am EST        September 26, 1987
*    spacestamp:    21E Halcyon Drive   West Yarmouth, Mass.   02673


*  name of the output file is rectCDEF.Rsrc, type is RSRC,
                                          creator is RSED
rectCDEF.Rsrc
RSRCRSED
```

```
*  grab code resource 1 and turn it into a purgeable control
                                                    definition

TYPE CDEF = PROC
,40 (32)
rectCDEF
```

End Listing Six

Listing Seven

```
*    this file is called rectCDEF.R1

*    ©1987 by Stan Krute   --   all rights reserved

*    timestamp:       12:41 am EST      September 26, 1987
*    spacestamp:      21E Halcyon Drive West Yarmouth, Mass.  02673


*   name of the input and output file is custom controls demo
!:custom controls demo

*   grab CDEF resource from rectCDEF.Rsrc
INCLUDE :rectCDEF.Rsrc
```

End Listing Seven

Listing Eight

```
*    this file is called rectCDEF.R2

*    ©1987 by Stan Krute   --   all rights reserved

*    timestamp:       12:41 am EST      September 26, 1987
*    spacestamp:      21E Halcyon Drive West Yarmouth, Mass.  02673


*   name of the input and output file is custom controls demo PROJ
.rsrc !:custom controls demo PROJ.rsrc

*   grab CDEF resource from rectCDEF.Rsrc
INCLUDE :rectCDEF.Rsrc
```

End Listings

STRUCTURED PROGRAMMING

Listing One *(Text begins on page 108.)*

```
Program maddints;

  { This program illustrates addition of two 180 x 180 integer  }
  { matrices A and B, storing the results in a similar matrix C }

Const  max = 180;       { maximum columns/rows per square array }

Type   arrayPtr = ^bigArray;
       bigArray = array [1..max, 1..max] of integer;

Var    A, B, C : arrayPtr;
       x, y    : word;
{ ------------------------------------------------------------- }

Procedure acquire (var D : arrayPtr);

  { Allocates the array on the heap and fills it with data      }
  { This is a sample procedure for demo only. Replace it with   }
  {  the requisite code to acquire data from an external source }

Var   r, c : word;

Begin
  New (D);                                        { allocate a node }
  For r := 1 to max do
    For c := 1 to max do
      D^ [r, c] := (r * 10) + c;  { value of each array element }
End;
{ --------------------------- }

Begin     { main program }
  Acquire (A);                                  { acquire array data }
  Acquire (B);
  New (C);                                { set aside space for result }
  For y := 1 to max do
    For x := 1 to max do
      C^ [y, x] := A^ [y, x] + B^ [y, x];   { add arrays into C }
  Writeln ('Proof:');
  Writeln ('A [1, 1] = ', A^ [1, 1]);
  Writeln ('B [1, 1] = ', B^ [1, 1]);
  Writeln ('C [1, 1] = ', C^ [1, 1]);
  Writeln;
  Writeln ('A [max, max] = ', A^ [max, max]);
  Writeln ('B [max, max] = ', B^ [max, max]);
  Writeln ('C [max, max] = ', C^ [max, max]);
End.
```

End Lising One

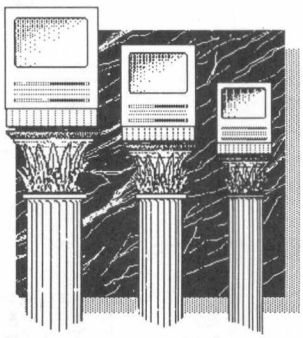
Listing Two

```
 Program hugemats;

   { Demo program to add two huge matrices > 64K, giving a third }

Const  maxRows = 250;
       maxCols = 300;

Type   dataObj = word;
       colPtr = ^colArray;
       colArray = array [1..maxCols] of dataObj;
       colNode = record
           col : colPtr;
         End;
       rowPtr = ^rowArray;
       rowArray = array [1..maxRows] of colNode;

Var    A, B, C : rowPtr;
       x, y    : word;
       error   : Boolean;
{ ------------------------------------------------------------- }
Procedure create (var D     : rowPtr;
                  var error : Boolean);

  { Create huge array 'D' and pass back pointer to it }

Var   row : word;

Begin
  Error := false;
  If maxAvail > sizeof (rowArray) then     { if space available }
    GetMem (D, sizeof (rowArray))          { allocate row array }
  Else begin
    D := nil;
    Error := true;
  End;
  If D <> nil then
    For row := 1 to maxRows do begin        { allocate all rows }
      If not error then
        If maxAvail > sizeof (colArray) then          { if space }
          GetMem (D^ [row].col, sizeof (colArray))  { alloc row }
        Else
          Error := true;
    End;
End;
{ --------------------------- }
Procedure acquire (var D     : rowPtr;
                   var error : Boolean);

  { Load data into array 'D' after creating it }

Var   row, c : word;

Begin
  Create (D, error);
  If not error then
    For row := 1 to maxRows do
      For c := 1 to maxCols do
        D^ [row].col^ [c] := (row * 10) + c;   { modify to suit }
End;
{ --------------------------- }

Begin   { main program }
  Writeln ('Size of each array is ',
           sizeof (rowArray) + (sizeof (colArray) * maxRows),
           ' bytes');
  Writeln ('Initial heap space = ', memAvail);
  Writeln ('Setting up array A');
  Acquire (A, error);

  If not error then begin
    Writeln ('Remaining heap space = ', memAvail : 6);
    Writeln ('Setting up array B');
    Acquire (B, error);
  End;
  If not error then begin
    Writeln ('Remaining heap space = ', memAvail : 6);
    Writeln ('Creating target array C');
    Create (C, error);
  End;
  If not error then
    Begin
      Writeln ('Remaining heap space = ', memAvail : 6);
      Writeln ('Adding arrays');
      For y := 1 to maxRows do
        For x := 1 to maxCols do
          C^[y].col^[x] := A^[y].col^[x] + B^[y].col^[x];

      Writeln;
      Writeln ('Proof:');
      Write   ('A [1, 1] + B [1, 1] = C [1, 1] = ');
      Writeln (A^[1].col^[1] : 5, ' + ', B^[1].col^[1] : 5, ' = ',
               C^[1].col^[1] : 5);
      x := maxCols;
      y := maxRows;
      Write   ('A [m, n] + B [m, n] = C [m, n] = ');
      Writeln (A^[y].col^[x] : 5, ' + ', B^[y].col^[x] : 5, ' = ',
               C^[y].col^[x] : 5);
    End
  Else
    Begin
      Writeln ('Out of memory: program ended');
      Write (#7);                                          { beep }
    End;
End.
```

End Listings

TO THE MACS

MultiFinder, HyperCard, and More on Custom CDEFs



In case you missed January's introductory column: I'm here to talk Mac. Mostly I'll provide code samples. There'll also be discussion of programming tools, notable applications, conversations with programmers, and interesting ROM/OS topics.

This month I discuss three goodies from Apple (MultiFinder, HyperCard, and a manual) and finish up the custom control definition (CDEF) code project I began last time. If you hadn't noticed, the Macintosh universe is going supernova. Time to surf the flash....

MultiFinder Arrives

Official production copies of MultiFinder flew into my mountain aerie this month. (Although you will read this in February, I wrote it in October/November.) Bottom line: I love it. A clean bit of work. Thanks, Apple people. I run it almost all the time. The exception, when I'm print spooling to my ImageWriter. That'll change, probably by the time you read this.

Meantime, a few thoughts:

- Two Mbyte becomes the new minimum Mac RAM configuration. Think of the momentum Apple'd have if this chip-war silliness weren't happening....
- There may finally be a blessed stop to "What would you do with (fill in the blank) Mbyte of memory, anyway?" questions, now that the obvious answer becomes even more so: Fill it with programs and data that work intelligently with one another so as to further enlarge the universe of possible computer users.
- The major hole in this first version of MultiFinder: interprocess communication facilities. To raise the intelligence of the computing environment, applications must be able to talk to one another, both demo- and autocratically. Apple seems to understand this, so I figure this feature will show up in the next release. Plan ahead.
- Desk accessories, though supported, have lost a big chunk of their prime raison d'etre: pseudomultitasking. We've now got semimultitasking, with the full thing lurking a few months ahead.

By the way, in case you don't know: with this release of MultiFinder, holding down the Option key when invoking a DA opens it in the foreground application's world rather than MultiFinder's separate desk accessory layer—useful for various parasitic DAs (spelling checkers, macro engines, et al.).

- The obvious hook into multitasking, GetNextEvent, has been used. If you guessed this three years ago, give yourself a no-prize.
- Following my favorite thread in the Apple mythos, MultiFinder favors decentralization of control. Rather than use a central multiprocessing dictator, MultiFinder gives applications the major say in CPU allocation. I like the implications.
- If you haven't already, say goodbye to low memory. Say goodbye to tricky OS hacks. Say goodbye to direct writing to the screen. Say goodbye to event and window manager fiddling. Assume as little as possible about anything. *Don't play no dirty tricks*.

Of course, capital-G Good Mac programmers never did funniness for fun, anyways. They did it to get around some performance hole in the environment. So, OK, we won't do that anymore. But now it's up to Apple to make sure the necessary functionality gets built into the official toolkit. And while you're at it, Cupertinians, how about a beefed-up Quickdraw/PostScript on a chip, please?

It's not too hard to get your programs "MultiFinder friendly." Call or write APDA for technical documentation, sample code, interfaces, and so on. I'll also try to work up something unique for the column. The docs I've got now are preliminary; by the time you read this, a lot more should be available.

HyperCard, Too

Another official arrival these past months: HyperCard. I like it a lot. We need a speedy and standard engine for accessing the gargantuan lased-out data piles that are looming, and HyperCard will do just fine.

HyperTalk, the HyperCard programming language, is simple, elegant, and powerful. Hits me as a nice synthesis of the history and state of computer languages, a sort of where-we-stand-today for the masses. I've done a fair amount of teaching programming, and this is a language I could use with all levels of learners.

There's a lot of the Atkinson snap to HyperCard, along with his attention to user interface and small implementation details. Quibbles? Well, there are the obvious things: You can only open a stack at a time. You can't resize the HyperCard window. There's a fair amount of modality and isolation from other applications. I have the feeling these things will get fixed.

Want to program the sucker? Play with it. Examine/modify scripts. Scan Apple's *HyperCard User Guide*. Then go on to Danny Goodman's fine work, *The Complete HyperCard Handbook* and APDA's HyperCard Techni-

by Stan Krute

(continued from page 90)

cal Reference Package. Also, Peter Olson, proprietor of Delphi's ICONtact Mac forum, has written a sweet piece of shareware, Stackware Detective, to let you peek a little deeper. Check it out, and remember to pay if you end up using it.

My own HyperCard explorations are traveling along the *XCMD* and *XFCN* paths. These are hooks that let you add your own commands to HyperTalk. You can find details in the APDA docs. I'll be bringing you some samples in the near future. As *PEEK* and *POKE* are to BASIC, and slots are to the Apple II, the *X* twins are to HyperCard.

One of the Great Docs

Apple's always impressed me with its documentation, especially the programmers' docs. I think of the early Applesoft manuals, the Apple Pascal books, *Inside Macintosh*, Bo Three Bob's tech notes, and on and on. There have always been individuals at the company who care about this sort of thing, realizing its importance, and they've been able to marshal resources and produce.

There's an especially nice document out that you may have missed. It's one that all Mac programmers should read, ponder, and pay attention to. It's title is *Human Interface Guidelines: The Apple Desktop Interface*, and it's available from APDA.

I know, the title's not particularly exciting. But the content is, to me at least. Some of the more frustrating discussions I've had are with programmers who just don't understand why they should follow Apple's interface guidelines as closely as makes sense for their application. It's not because all Apple's decisions are cosmically correct. They're not. But they're darn close. And above all quibbling, consistent operational similarities between applications empower users. Period.

Please read this book, folks.

Code Corner: Custom CDEFs, Part 2

Resource Revelations

Minor problem: As noted last month, I use ResEdit to build most of my Macintosh program resources. It's so Mac-like. But there's no text file script left over. So how can I reveal custom controls demo's resources to you?

Macintosh Programmer's Workshop (MPW) includes a Mac resource decompiler, DeRez. It takes a file's resources and produces a text file script description. It'd do the trick. But I don't use MPW (yet?). Cruising Delphi, my favorite on-line hangout, I found a fine solution: Alan Dahlbom's ResTools 2.01. It does much of what MPW's Rez and DeRez do, uses a compatible C-like syntax, and is free. A big thanks to Alan for a fine piece of programming and a public service.

So, Listing One, starting on page 64, shows the ResTools 2.01 decompilation of most of the resources included in custom controls demo PROJ.rsrc. The syntax, as mentioned earlier, is C-like; if it doesn't make immediate sense, go to *Inside Mac* and stare at the particular resource's description.

I've removed the decompilation of the file's *PICT*s, *ICON*s, and *ICN#*s; these graphic resources look kind of silly as streams of hexits. Table 1, below, indicates where you can find images of the *PICT*s as well as the

size of their bounding rectangles. Figure 1, also below, shows some of the *PICT* images. Figure 2, page 100, shows the *ICON*s and *ICN#*s. I also removed the CDEF from the decompilation (more meaningless hex). It's produced by compiling rectCDEF.Asm, as described later.

Lightspeed C takes the resources in custom controls demo PROJ.rsrc and binds them into the finished custom controls demo application. Other Mac development systems let you perform congruent conjunctions.

Custom Controls Demo's Resources

BNDL 128—This BuNDLe resource connects the program to its Finder icon.
CDEF 40—The Control Definition code.
CNTLs 1 thru 21—CoNTroL specifications for each of the program's main modal dialog's 21 custom control items.
CuCo 0—The application's version data/signature/creator resource: a Pascal-type string providing its name, version, and date.
DITL 1—DIalog ITem List for the program's main modal dialog.
DITL 210—DIalog ITem List for the copyright button's modal dialog.
DLOG 1—DiaLOG template for the program's main modal dialog.
DLOG 210—DiaLOG template for the copyright button's modal dialog.
FREF 128—File REFerence that hooks the program up with its Finder icon.
ICN# 128—The program's Finder icon.
ICONs 30, 110, 120, 121, 130, 140, 141, 150, 160, and 161—ICONs used with various control items in the main modal dialog. Numbering convention: the control's item number in the dialog times 10, with 1 added for a content-change image.
MENU 1—Menu template for the program's menu.
PICTs 50, 51, 60, 61, 70, 80, 90, 100, 101, 170, 171, 200, and 201—PICTures used with various control items in the main modal dialog. Numbering convention: the control's item number in the dialog times 10, with 1 added for a content-change image.
PICT 210—PICTure used with the copyright button's modal dialog.
STRs 20 and 40—STRings used for content changing with the modal dialog's control items 2 and 4.

PICTs 50 and 51—see last month's Figure 3, variation 5—**rect: 0,0,57,61**

PICTs 60 and 61—see last month's Figure 3, variation 7—**rect: 0,0,68,69**

PICT 70—see last month's Figures 4 and 7, DITL item 7 —**rect: 0,0,34,32**

PICT 80—see last month's Figures 4 and 7, DITL item 8 —**rect: 0,3,196,116**

PICT 90—see last month's Figures 4 and 7, DITL item 9 —**rect:0,12,35,73**

PICTs 100 and 101—see last month's Figure 3, variation 9—**rect: 0,0,56,70**

PICTSs 170 and 171—see last month's Figures 4 and 7, DITL item 17, and this month's Figure 2—**rect: 0,0,187,85**

PICTSs 200 and 201—see last month's Figures 4 and 7, DITL item 20, and this month's Figure 2—**rect: 0,0,24,24**

PICT 210—see last month's Figure 2—**rect: 1,14,168,290**

Table 1: *Finding the* PICTs*, and their bounding rectangles, from custom controls demo*

Figure 1: *Selected* PICTs *from custom controls demo*

CDEF Development Files

As mentioned in last column, the CDEF is written in assembly language, using MDS 2.1. *Figure 3, page* 100, shows the files involved in its development. rectCDEF.Asm and rectCDEFEqu.Txt are the critical files; the others are either MDS-specific or adjuncts to development. Here are brief descriptions:

rectCDEF.Job—An MDS executive file used to control one iteration of the development cycle. After compiling the CDEF, it turns it into a code resource, puts the resource into several files, and ends up in custom controls demo, ready for testing. See Listing Two, page 65.
rectCDEF.Asm—Assembly-language source code for the CDEF. Entry point is at the beginning of the code. See Listing Three, page 66.
rectCDEFEqu.Txt—Private definitions for rectCDEF.Asm. See Listing Four, page 83.
rectCDEF.Link—Instructions for the MDS linker. Takes the .Rel file produced by assembling rectCDEF.Asm and produces a standard Mac CODE

TO THE MACS
(continued from page 98)

resource. See Listing Five, page 83.
rectCDEF.R0—Instructions for the MDS resource compiler. Takes the standard CODE resource produced by the linker and turns it into a CDEF. See Listing Six, page 84.
rectCDEF.R1—Instructions for the MDS resource compiler. Takes the CDEF produced via rectCDEF.R0 and adds it to custom controls demo. See Listing Seven, page 84.
rectCDEF.R2—Instructions for the MDS resource compiler. Takes the CDEF produced via rectCDEF.R0 and adds it to custom controls demo PROJ.rsrc. See Listing Eight, page 84.
rectCDEF.Rel—The .Rel file produced by assembling rectCDEF.Asm.
rectCDEF—The file produced by linking rectCDEF.Rel under the control of rectCDEF.Link.
rectCDEF.Rsrc—The file produced by the MDS resource compiler that contains the CDEF resource.
custom controls demo—My application, to which the MDS resource compiler adds the CDEF for easy testing during CDEF development.
custom controls demo PROJ.rsrc—My application's resources, to which the MDS resource compiler adds the CDEF.

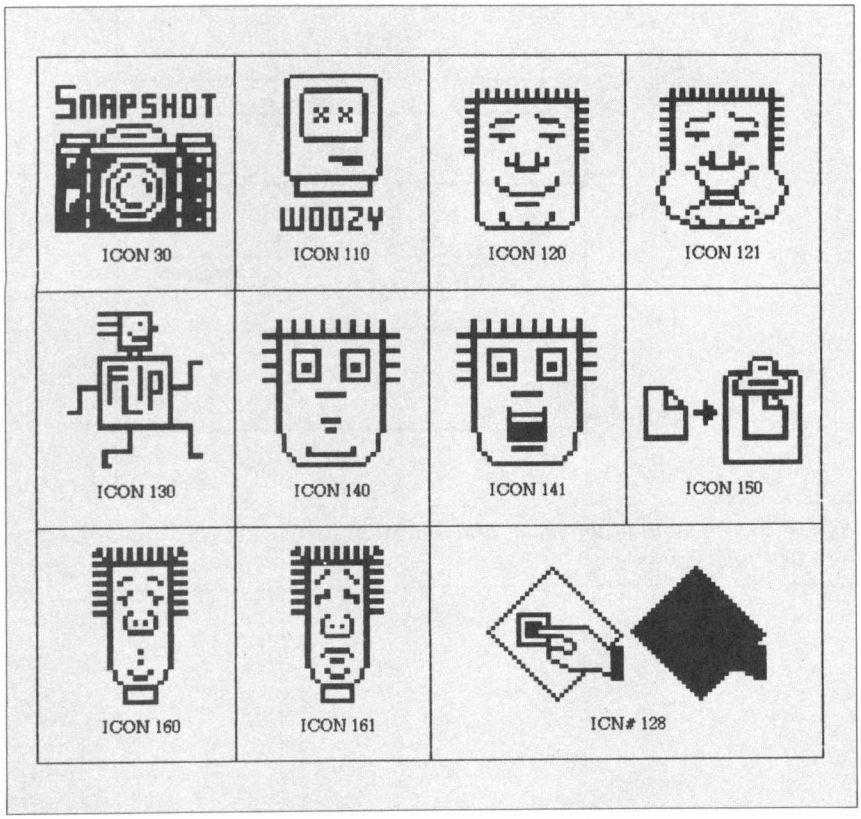


Figure 2: ICONs *and* ICN#s *from custom controls demo*

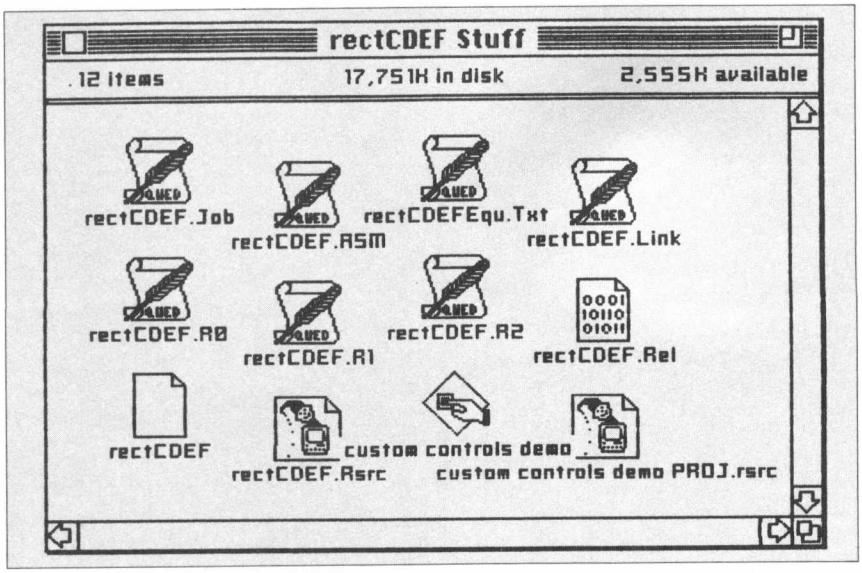


Figure 3: *The* rectCDEF *files*

CDEF Refresher

A C function prototype for a CDEF looks like this:

```
pascal long someCDEF ( int varCode,
ControlHandle theControl, int mes-
sage, long param ) ;
```

varCode indicates which of the CDEF's variations to use; *theControl* gives you a handle to the calling control's control record; and *message* tells the CDEF what to do. There are nine possible messages—see *Inside Macintosh* and/or the message jump table in the source code. Finally, *param* is 4 byte of data that vary according to the message passed to the function. The meaning of the function result also varies with the message; the default return value is 0.

The *pascal* keyword in the prototype indicates that the function is called with Pascal calling conventions. In rectCDEF.Asm, I use *LINK/UNLK* to create space for some local variables. Figure 4, page 101, shows how the stack looks after the CDEF is called and the *LINK A6* command's been given.

The Clean Nut Can't Shut Up

As I never tire of pointing out: good Mac programming has to play by the rules. Keep it clean, and your programs will run now and in the future. What entaileth clean? Deal with error codes. Lock down heap objects only when you must, and let them float whenever you can. Assume nothing. Steer clear of low memory. Follow *Inside Mac* and Apple's *Mac Tech Notes* religiously. Etc., etc., etc.

Gawd, I sound like the Ms. Grundy of cybernetics. But, hey,

cleanliness works. Clean Mac programs I wrote three years ago function on Mac IIs under MultiFinder.

Take a look at the *rectCDEF* code, for example. The control that's calling the CDEF gets its control record locked down during any call to the CDEF (see the CDEF's entry area). The same with the control's auxiliary data block (also in the CDEF's entry area). Both of these data objects are released at the end of the CDEF call. Calls to the resource and memory managers (in *doInitCntl*) are carefully checked for success/failure, and the program tries to respond reasonably to the results (see *doInitCntl* and *doDispCntl*).

This is a 680x0 we've got, with all those lovely registers waiting to serve. So pack 'em up. That's what I try to do in the CDEF. Here's the scoop: Three address registers are taken up right off the bat. *A7* points to the stack, *A6* points to a stack frame, and *A5* fingers application and *Quickdraw* globals. Then, I use *A2* to point to the control's record and *A3* to point to its auxiliary data block. That leaves *A4* free for temporary usage and *A0* and *A1* for even more temporary usage. That's because ROM/OS calls generally trash *A0* and *A1*.

Data registers next: I use *D3* to hold the function parameter *param*. *D4* holds flags that let me quickly scoot around the code based on which of the 16 CDEF variations is in effect. That leaves *D5*, *D6*, and *D7* free for temporary usage and *D0*, *D1*, and *D2* for even more temporary usage. Again, that's because ROM/OS calls can trash *D0*, *D1*, and *D2*.

On entry to the CDEF, I' build a stack frame, save registers that I'll use, grab some parameters, grab a set of variation flags, set a default function result, lock the control record down, point to any auxiliary data block and lock it down (if it exists), then jump to a specific routine based on the function's message parameter.

Returning from that specific routine, I make a somewhat symmetric exit: unlock any auxiliary data block, unlock the control record, restore saved registers, remove the stack frame, and jump on back to the CDEF's caller.

The CDEF handles 16 control variations. I use an 8-bit flag to signal eight qualities of a given control variation. This simplifies the rest of the CDEF function's control logic. Take a look at *varFlagTable* for details.

Handling Controls

doInitCntl takes care of any special initialization needs. For this CDEF, I need an auxiliary data block to hold handles to variant-specific resources.

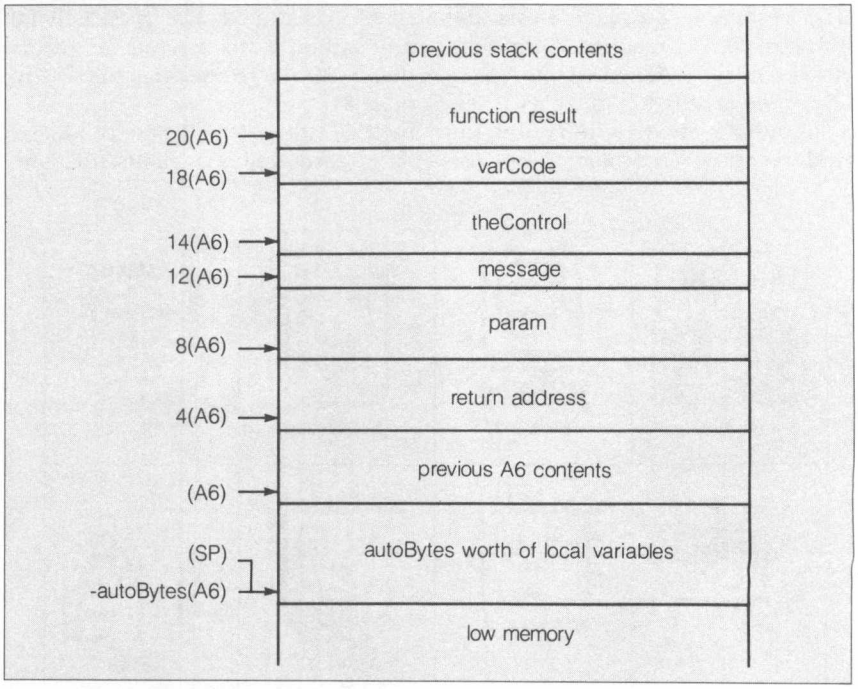


Figure 4: *The state of the stack upon entry to* rectCDEFProc, *just after the* LINK *instruction*

TO THE MACS
(continued from page 101)

So *doInitCntl* first tries to get a block. If it succeeds, it then looks to see what resources are needed, tries to load them in, and then stores the resulting handles in the auxiliary data block.

Note the code's provisions for failure. No memory available for the auxiliary data block? You jump nimbly out of the initialization, leaving a tell-tale *NIL* handle behind in the control record. Resources can't be loaded? Again, *NIL* handles are stored to tell the tale. Other routines in the CDEF check for all these *NIL* handles and and react appropriately.

doDispCntl takes care of any special control disposal operations. In this case, I need to release any variant-specific resources that were loaded during initialization, then get rid of any auxiliary data block. The code's pretty straightforward. Note how I check for, and dance around, any *NIL* pointers/handles.

doDrawCntl is invoked when the CDEF's asked to draw the control. It first saves grafport features it may change. Then it cases out on the type of outline the control needs: shadowed, simple, or none. After drawing the outline, it sets the grafport's clipping region to the control's interior. Then there's another case-out, this time to draw the button's interior, which can contain text, icon, or a picture. After the interior's drawn, the routine restores the saved grafport features and ends.

Interior clip regions make sure that button contents don't spill out of bounds.

Outlines and Interior Clip Regions

It's fairly simple stuff, really. *rectCDEF* supports two styles of outlining: shadowed and simple. *doShadOutline* draws shadowed outlines. First it draws two shadow lines, then a rectangle. *doSimpOutline* draws simple outlines by drawing a rectangle.

Interior clip regions make sure that button contents don't spill out of bounds. For outlined buttons, you just take a rectangle slightly inset from the button's bounding rectangle. Naked buttons use the bounding rectangle.

Drawing Button Interiors

doTextInterior draws text button interiors. It starts by figuring out which font the button's text will get drawn in and sets the grafport accordingly. Then it uses the font information to figure out vertical positioning for the text.

Next, the routine gets a pointer to the text string it'll draw. This can be the control's title or, in the case of a content-changing button, a *STR* resource. If it's a STR resource, the resource gets locked down. The routine uses the text string to figure out horizontal positioning. Now that the vertical and horizontal positions are known, *doTextInterior* moves the grafport's drawing pen into place to start drawing the string.

Depending on the button's type and state, the interior is painted white or black. Based on the same information, the routine draws the text string in black or white. If a *STR* resource supplied the text, it's now unlocked. And, if the button's in an inactive state, black text gets turned to gray. The routine restores the grafport and exits.

doPictInterior draws picture button interiors. First it figures out the *PICT* resource to use, then it locks the *PICT* down. It uses the *PICT*'s height and width to set up a destination rectangle that'll center the picture in the button's interior. If the picture's too big, the destination rectangle is set so its upper-left corner matches the button interior's upper-left corner. Then the routine erases the button's interior and draws the *PICT*, and the *PICT* gets unlocked. Finally, a separate routine, *interiorAdjust*, is called to deal with any necessary image inverting or graying out.

Very similarly, *doIconInterior* draws iconic button interiors. It uses the standard height and width of an icon to set up a destination rectangle that'll center the icon in the button's interior. If the icon's too big, the destination rectangle is set so its upper-left corner matches the button interior's upper-left corner. Then the routine erases the button's interior and draws the icon. Finally, *interiorAdjust* is called to deal with any necessary image inverting or graying out.

Procedural wrap up

doTestCntl tests a point to see if it's contained within an active control. If it is, the routine sets the CDEF's function result to the control's part number. If the point isn't in the control, or if the control's inactive, the routine sets an appropriate result code.

Because my controls are rectangular, the containment test is simple: just call on the Mac's *PtInRect* function, which checks for a point's containment within a rectangle.

DoCalcCCntl is deliciously trivial. Somebody wants the control described as a region? You just send the control's rectangle to the Mac's *RectRgn* procedure, which takes a rectangle and produces a corresponding region description.

There are four messages to the CDEF that you ignore, relying instead on the Mac's default behaviors. So *doPosCntl*, *doThumbCntl*, *doDragCntl*, and *doAutoTrack* are just stubs.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Bibliography

Apple Computer Inc. Human Interface Guidelines: The Apple Desktop Interface. Human Interface and Technical Publications Groups APDA #KNBHIG.

Goodman, Danny. The Complete HyperCard Handbook. : Bantam, 1987.

MultiFinder Development Package Version 1.0. Available from APDA #KMSMSD, $12.50.

DDJ

(Listings begin on page 64.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Vendors

APDA
(Apple Programmer's and Developer's Assoc.)
290 S.W. 43rd St.
Renton, WA 98055
(800) 426-3667
In Wash. (800) 527-7562
In Canada (800) 237-4644
(206) 251-5222

HyperCard
Available from Apple dealers

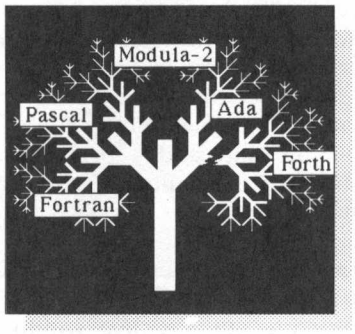
HyperCard Technical Reference Package
APDA #KMBHTL
Available from APDA
(see above)

MultiFinder 1.0
(plus System 4.2 and Finder 6.0)
Available from Apple dealers

ResTools 2.01
Alan Dahlbom
Downloadable from Delphi, GENie, and other on-line services

Stack Detective
Peter Olson
Shareware available from Delphi and other shareware sources

Handling Huge Arrays



Let's face it, when IBM selected the 8088 as the engine for its first-generation PC, memory-segmented architecture officially became a bad idea whose time had come. Three-and-a-half generations of Intel chips later, we're still stuck with it, and there's no relief in sight. Credit for all the remarkable things done on PCs goes not to the architecture but to those with the ingenuity to find ways around the flaw in its addressing scheme: no single hunk of memory in a PC can be bigger than 64K.

Compilers for the PC have dealt poorly with the problem. You can have multiple code segments and in some cases multiple data segments, but the 64K limit still applies to any single extent. Probably someone will write to tell me that the latest release of Googlephonic Snobol or some such has made segmentation transparent, but I'm talking about the mainstream languages from the likes of Microsoft and Borland. They deal with memory segmentation by not dealing with it in any graceful fashion.

The solution is, of course, to keep all objects (code units, global variable sets, arrays, and so on) smaller than 64K. This is usually a workable solution because most objects are inherently smaller than the magic number. In some cases, though, it's simply not possible to remain within this arbitrary limit. Nobody gets hit harder in that regard than scientists and engineers, who routinely work

by Kent Porter

with extremely large numeric arrays.

Many have tried the PC and found segmentation an insuperable barrier, even though their data would, given a decent addressing scheme, fit within the free memory of the average PC. This article is for them and for anyone else faced with the unsavory task of attempting to process arrays of several thousand elements.

Memory Segmentation

Memory segmentation works like this: It takes two elements to express a complete address. The first element is a base, or point of reference, and the second is an offset from the base. In the Intel chips, the segment registers (*CS*, *DS*, and *ES*) contain base addresses that remain relatively fixed so that the offsets of variables, entry points, and so on can be reliably measured from them.

A segment address is actually the upper four digits of a five-digit hex number. Therefore, each increment of the segment represents a 16-byte jump in real memory. In the endless quest for quaint terminology, someone has dubbed these 16-byte hunks "paragraphs." A segment always begins at a paragraph boundary, which is a multiple of 16 bytes.

The offset is another four-digit hex value that measures a distance in bytes toward the top of memory, using the segment as the reference point. Four hex digits, of course, can represent any of 64K values. Consequently, any addressing range in the PC is constrained to 64K from a paragraph boundary.

The *CS* register contains the base address for a code segment. When a program takes a long jump (or far call), the *CS* register changes to the base paragraph of the new code segment. A far return reinstates the old *CS* so that execution resumes in the original code space. Thus a program can have multiple code segments, each of up to 64K.

The *DS* (data segment) register is less flexible. Compilers usually enforce a stable *DS* throughout program execution so that globals are universally accessible, meaning that the total size of all globals cannot exceed 64K, lest they be beyond the range of the offset.

Auto variables—those allocated as locals within subprograms or passed as parameters—are even more constrained. They're allocated on the stack—another 64K-max structure—so that space is limited to 64K less the stack space already occupied by other stuff.

So what's a body to do with a huge hunk of data? There are a couple of alternatives. One is to build and process array images on disk. That's an ugly way out. You'd better start a big matrix multiplication just before a long weekend. Another option, which I'll explore here, is using the heap.

Purists may quibble, but the heap is essentially all the memory that's left after any software currently in memory has staked its claims. Unless specifically set up otherwise, .COM and .EXE programs consider the whole of uncommitted memory as their heap space. Given a moderately sized program and a couple of memory-resident utilities, a 640K machine can usually yield a heap of about 450K to 500K. That's a repectable amount of space: a quarter of a million integers, or upward of 60,000 to 80,000 reals depending on precision. The trick is to use it effectively.

What Can You Put into 64K?

Programming strategies for managing huge arrays inevitably involve some pencil sharpening. Let's first consider how much data can fit inside a 64K node allocated on the heap.

Compilers typically claim a few bytes of memory for record keeping.

(continued from page 108)

For a large array occupying a heap node, figure the overhead at 16 bytes, which leaves 65,521 for data. Divide that by the size of the array's data type to find out how many elements will fit, truncating any fractional result. Table 1, page 111, lists the sizes of some common data types.

Taking Turbo Pascal reals as an example, 65,521/6 = 10,920 array elements. That's an approximately square matrix of 104 × 105, or one of 26 × 420, or any other combination of multiples and submultiples whose product is 10,920 or less. Similarly, a 64K array of IEEE doubles can have 8,190 elements, or 90 × 91, 182 × 45, 30 × 273, and so on.

Assuming that's enough for your matrix, it's easy to make the necessary declarations and allocate the space. You might define the array in Pascal as:

```
TYPE arrayPtr = ^bigArray;
  bigArray = ARRAY [1..90, 1..91]
OF DOUBLE;
```

and declare a pointer variable as:

```
VAR arr1 : arrayPtr;
```

Allocation is then accomplished with the standard procedure *NEW (arr1);*, after which you can refer to elements with the notation *arr1^ [row, col]*.

Pascal/Modula-2 pointer types such as *arrayPtr* are automatically 32-bit *segment:offset* objects. You can therefore allocate several 64K arrays on the heap and, using subscripted pointer notation as above, compare and combine them.

Listing One, page 86, illustrates this with a program named maddints.pas. It adds two integer matrices, each 180 × 180, and stores the results in a third heap node. An array of 180 × 180 encompasses 32,400 elements, or 64,800 bytes. There are three such arrays, occupying a total of 194,400 bytes of heap space.

Maddints is somewhat artificial in that it sets both *addend* arrays to the same values, where each element contains a quantity computed as:

```
D^ [row, col] := (row * 10) +
column;
```

Thus the elements of the first row of both *A* and *B* are 11, 12, 13, . . ., of the second 21, 22, 23, . . ., and so on, and the elements of the sum array *C* are double those of the sources (22, 24, 26, . . .). To make this into a real-world program, revise the *Acquire* procedure to read the data from an external source such as a disk file and add a procedure to output the results to a suitable medium.

Matrices Bigger Than 64K?

Truly huge matrices require a bit more ingenuity to live within the means of 64K segments. In this case, each row of the matrix is a separate array of up to 64K in size, with the subscripted elements representing columns. A matrix of *n* rows consists of *n* such horizontal lists. The glue that holds it together is another array—a vertical list containing n pointers to the rows in sequence. Figure 1, this page, illustrates this structure, where *D* points to an array of pointers, each of which points to an individual row.

Conceptually, the matrix elements can be regarded as D [1..n, 1..z], where n is the number of rows and z the number of columns. Because of pointer following, however, the actual Pascal/Modula-2 notation is more complex. The next couple of paragraphs develop the notation based on a model.

You can define the types for this matrix structure as:

```
TYPE dataObj = WORD; {or whatever type is appropriate}
  colPtr = ^colArray; { pointer to a row }
  colArray = ARRAY [1..maxCols]
OF dataObj; { row }
  colNode = RECORD
        col : colPtr; { row pointer node }
      END;
  rowPtr = ^rowArray;
  rowArray = ARRAY [1..maxRows]
OF colNode; { list }
```

The sanity of defining the *colNode* record with only one field will become apparent when you see the actual notation. Also, it seems contradictory to use the names *colPtr*, *colArray*, and so on in referring to a row, until you realize that you use these objects to identify a specific column within a given row using subscripts.

The variables declared from these type definitions are very simple:

```
VAR A, B, C : rowPtr;
```

That is, the only variables deriving from the types are pointers to the controlling lists for the three matrices *A*, *B*, and *C*. After creating the matrices, you can refer to individual elements by following the pointer structure. For example:

```
A^ [25].col^ [287]
```

refers to the element in row 25,

Type	Bytes
Char, byte	1
Integer, word	2
Long integer	4
Pointer	4
Turbo Pascal real	6
IEEE single	4
IEEE double	8
IEEE extended	10
IEEE comp	8

Table 1: *Some common data types and their storage sizes*

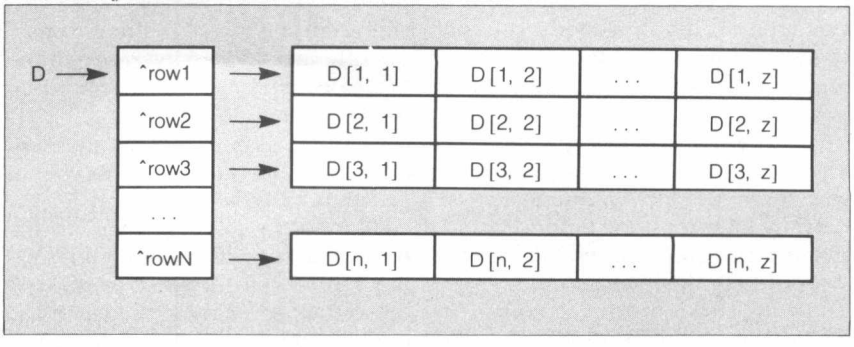


Figure 1: *Structure of a huge array*

column 287.

An ever-present danger when working with huge arrays is that you'll run out of heap space. Most languages furnish an intrinsic function to inquire about space availability. Logitech Modula-2, for example, has an *AVAILABLE* function that returns a Boolean indicating if the requested number of bytes can be allocated; Turbo Pascal 4.0 has *maxAvail*, which returns the size of the largest contiguous block. Before attempting to allocate space for a huge array, inquire whether your request will be honored. If it won't be, terminate gracefully. This is preferable to simply letting the program crash with some arcane run-time error message.

Listing Two, page 86, is a program hugemats.pas, written in Turbo Pascal 4.0, that implements this discussion. It's similar to Listing One, except that the matrices are 250 rows by 300 columns, or 75,000 elements apiece. Because the data object is of type *WORD* (an unsigned integer), this comes to 150,000 byte per matrix for data, plus 1,000 byte for the 250 row pointers, for a total of 151,000 byte. There are three such matrices, claiming 453,000 byte of heap space.

That's too much data for the program to run in the Turbo Pascal interactive environment, which provides less than 300K of heap space on a 640K machine. Because of the space checking in the *create* procedure, the program will always terminate with an out-of-memory message if the Turbo Pascal environment is resident. On my machine, the compiled .EXE program running alone has only about 20K of heap space left after the three matrices are created, but that's enough for it to run.

Freeing Up More Space

Huge number crunchers such as those in Listings One and Two are typically bare-bones programs, avoiding slick user interfaces and other memory-gobblers. Remember, every little nicety in a program consumes precious memory and eats into the heap. If you're tight on space, cut

STRUCTURED PROGRAMMING

out the gimmicks.

The stack is another place you can mine for memory. In Turbo Pascal 4.0, the default stack size is 8,192 byte; it's 8,000 byte in Logitech Modula-2. That's a waste of memory for programs such as those given here, which never have more than about 20 byte on the stack at once. Trim the stack down to a minimal size. The savings go into the heap, where you can use them productively.

Don't forget to free up dynamic space that's no longer needed. For structures such as those given in Listing One, you can loop through *rowArray*, calling *freeMem* in Turbo Pascal (or its equivalent in other dialects) to dispose of allocated nodes, and then free *rowArray* itself.

If your data requirements are still too huge to fit, you'll either have to give up on the PC or else resort to memory-stretching technologies such as EMS (which I'll discuss in a future column). The techniques shown here, however, should satisfy most requirements for working with huge arrays on the PC.

The suggestion for this month's column came from Dr. Don Walters, Professor of Physics at the U.S. Naval Postgraduate School in Monterey, Calif. If you have a programming issue you'd like to see addressed here, drop me a line (no calls, please) in care of DDJ. Or send an MCI message to KPORTER, Mountain View, Calif. No promises, but I'll tackle as many as I can.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 86.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

EXAMINING ROOM

Turbo Pascal, Version 4.0

Target:
IBM PS/2, PC AT, and true compatibles
Requires:
DOS 2.0 or later; 384K for the integrated environment or 256K for the command-line environment
Price:
$99.95
Vendor:
Borland International, 4585 Scotts Valley Dr., Scotts Valley, CA 95066; (408) 438-8400

With Version 4.0, Borland has turned its highly successful Turbo Pascal into a language package well suited to serious software development. Until now, Turbo Pascal has been a structured language suited for small, individually written projects. Previous versions have had certain limitations that barred them from consideration for major software projects. With Version 4.0, Borland has given us a Pascal that is still every bit as useful to its original market and that is also a worthy compiler for commercial and academic programmers.

Turbo Pascal 4.0 offers several new enhancements both to the Pascal language and to its programming environment. One example, the graphics unit, furnishes the most comprehensive set of screen-handling routines that I've seen with any general-purpose programming language. I'll discuss it presently, but first I'll look at some of the particulars.

Turbo Pascal 4.0 corrects the following oft-heard complaints regarding earlier versions:

- Each code and data segment limited to 64K: Version 4.0 code can be of any size up to the total of available memory. The data segment is still limited to 64K of globals, as it is with most other compilers for the PC, but as with earlier releases, you can overcome the 64K data limit by using dynamic allocation.

- No modular compilation: Version 4.0 encourages programmers to use separately compiled units. Programs thus become collections of precompiled modules (.TPU files) glued together by a main program.

- No object libraries: Version 4.0 furnishes utilities for managing compiled object libraries.

- Produces only .COM files: Version 4.0 produces .EXE files, and no separate link step is required. For equivalent programs compiled with other releases, the .EXE file is considerably smaller.

- Inserts unnecessary code into executable files: Version 4.0 optimizes the code, removing unnecessary instructions and unused routines.

- Not compatible with standard Pascal: Version 4.0 is almost compatible with the ANSI/IEEE770X3.97-1983 standard. The well-documented list of exceptions runs at less than two pages, many of them overriding counterproductive requirements of the standard. For example, the standard forces a program to abnormally end (abend) if there is no clause for the selector's current value in a *CASE* statement; Turbo Pascal 4.0 just falls through the *CASE* unless there's an *ELSE* clause.

Three levels of compilation are available. At the lowest level is a simple compile of the current unit. Next higher is *Make*, which recompiles the current program and any units that have been changed since the last compile. At the highest level is *Build*, which unconditionally compiles all the parts of an application.

The complete user interface is awakened by the *TURBO* command, which starts TURBO.EXE running. For hairy-chested traditionalists, a command-line compiler (TPC.EXE) is also available, replete with Unix-like switches. I can't imagine why anyone would prefer TPC; it doesn't furnish nearly as much functionality (although the language is exactly equivalent), and command-line compilers aren't much fun.

Whichever, compile speed is blazingly fast. Borland claims 27,000 lines per minute on an 8-MHz AT, which is the machine I have. Although I'm unwilling to write 27,000 lines of code simply to prove the company right or wrong, I'll say this: a 600-line program compiles fully before my finger leaves the command key. Even on an old 4.77-MHz XT, Turbo Pascal 4.0 compiles equivalent code as or more rapidly than Turbo C on the AT. And Turbo C is no slouch.

Linking

Both the integrated and command-line compilers take the somewhat unusual approach of incorporating the linker. This substantially decreases the overall cycle time from .PAS to resulting .EXE. The built-in linker brings in .OBJ files in standard Intel relocatable format; it iden-

Ron Copeland, associate editor, is the coordinator for this review section. He welcomes your feedback on products worth reviewing.

(continued from page 116)

tifies them via the *$L* switch, which names the .OBJ file. This allows you to link with modules written in assembly language or in C using the Pascal calling conventions.

Oddly, the compiler doesn't convert Pascal source files into .OBJ format. Source files beginning with the *UNIT* keyword become .TPU-compiled files, and those starting with the *PROGRAM* keyword become .EXEs. The linker automatically merges any .TPU units into the final product, identifying them from the *USES* statement.

For backward compatibility, Version 4.0 offers two units called Turbo3 and Graph3

Although this unconventional approach is valid most of the time, it makes an assumption that might not be universally true: that you want only to link foreign-language modules into Pascal programs and not the other way around. The absence of .OBJ output files and of any option to create them makes it impossible to export compiled Turbo Pascal modules to other languages. In effect, the built-in linker creates a one-way street leading from other languages to Turbo Pascal. You could wish it were a two-way street.

Compatibility with Version 3.0

Speaking of two-way streets, Version 4.0 does better with regard to Turbo Pascal 3.0 compatibility. Both upward and downward options are available.

A utility program called UPGRADE, distributed with Version 4.0, processes Version 3.0 source files in several different ways. One way marks up the original source file, pointing out obsolete Version 3.0 conventions and explaining what needs to be done. For experienced 3.0 programmers, this is a good way to find out where the mosquitoes are hiding, waiting to bite. Another method merges and unifies the source files for overlays and their parent in order to create a monolithic (multiple code segment) application. There are other options as well. UPGRADE is a forward-compatibility tool.

For backward compatibility, Version 4.0 offers two units called Turbo3 and Graph3. These allow a Version 4.0 program to employ 3.0 features such as turtle graphics, which Version 4.0 doesn't support.

The manual has a 20-page chapter devoted to converting from Version 3.0 to 4.0 in addition to a 12-page appendix detailing the differences between the two. That ought to give you some idea of the extent to which Turbo Pascal has been expanded and enhanced in Version 4.0.

Memory Management

While allowing executable code of any size, Version 4.0 is, of course, subject to the 64K segmentation imposed by the Intel processors. It gets around this by limiting the code in any compile unit to 64K and starting a new code segment for each module during linkage. Thus, intrasegment calls (to local subroutines) are near calls employing 16-bit offset pointers and those to external routines and to subprograms named in the interface portion of units are implicitly far (32-bit segment:offset) calls. Programmers have no control over these compiler-generated calling conventions.

An option lets you set the stack size to something besides 8K

You can, however, force a routine—or an entire compile unit—into far mode with the *$F* switch, in which *$F+* turns on far calls/returns and *$F–* reverts to default (compiler-controlled) calling conventions according to the rules above. An example of needing a far routine is when furnishing a mouse event handler: the handler is called as a far procedure from the device driver in low memory and thus must return with *RETF*. If the procedure heading is surrounded by the *$F* switches, its entire body is forced to far.

The *DS* register holds constant throughout the code, no matter how many code segments there are. Consequently, global data is limited to 64K, and references to global variables are always 16-bit offsets relative to *DS*. As in most high-level languages (and identically to Version 3.0), local variables for subprograms are allocated on the stack relative to the *BP* register and so do not count against the 64K limit for globals. Pointers to dynamic objects allocated on the heap are 32-bit segment:offset variables, allowing them to be passed freely among various code segments.

The default stack for a 4.0 program is 8K, and the heap is all of uncommitted memory above the stack. This eliminates the need to be concerned (as you were in Version 3.0) about heap/stack collisions, but of course it's possible for a highly recursive program to drive the stack down into the data segment. An option lets you set the stack size to something besides 8K and also to claim a finite area for the heap. The latter is important in multitasking and TSRs, neither of which can assume that they own all of memory.

An important addition to the Version 4.0 arsenal is a special interrupt-class procedure, which allows you to embed custom interrupt-service routines into your applications. These can be chaining routines ("hooks") that intervene in standard system interrupts, replacements for such things as the critical error handler, or new software- and hardware-driven routines activated via uncommitted vectors. An interrupt procedure looks much like a normal Pascal procedure, except that the *INTERRUPT* keyword follows the heading. Also, the heading can name the CPU registers you want preserved on the stack (subject to sequencing rules), and you can then read and write them as local variables.

A particularly nice feature is that the entry processing points the *DS* register to your program's data segment, furnishing direct access to all the program's globals. Because an interrupt procedure returns via an *IRET* after reloading the parametric registers from the stack, you cannot call it like a normal subroutine from within your program. That's as it should be. This feature alone makes Turbo Pascal, Version 4.0, a worthwhile compiler for serious developers.

Graphics

Another enhancement that scores high on the programmer's Richter scale is a new graphics unit. Encompassing some 60 functions and procedures, it provides a comprehensive set of primitives for controlling the display. These include optional automatic detection of the system's video capabilities and selection of the "best possible" mode among CGA, EGA in its many incarnations, VGA, Hercules, AT&T, and PC3270. You can also inquire about the display options available and force a particular mode selection.

Once in a graphics mode, the graph unit provides various line styles and widths as well as drawing and filling routines for common visual objects (bars, circles, polygons, rectangles, and so on). You can partition off viewports, set clipping boundaries, inquire about the screen's aspect ratio, monitor error status, and perform a host of other useful graphics operations. The graph unit supports *bitblt* operations for high-speed image transfers, including *XOR*, *OR*, *NOT*, and *AND*. This is a substantial improvement over Version 3.0's *PutPic/GetPic* operations, providing for much more sophisticated animation effects.

Also furnished with the graph unit are four stroked fonts: triplex, small, sans serif, and gothic. Unlike conventional bit-mapped character fonts, stroked fonts allow for justified and variable text sizes up to 10× that can originate at any pixel (rather than character cell) position and can be oriented either horizontally or vertically.

By default—in both text and graphics modes—Version 4.0 writes directly to screen memory rather than going through conventional DOS/ROM BIOS calls. This makes for extremely fast output. You can override the default, and a *CheckSnow* procedure is thoughtfully provided to prevent the unsightly "snow" occurring on older CGAs during direct screen writes. Nevertheless, the line-drawing routine is not particularly fast.

The 4.0 graph unit is impressive, providing the kind of display control and intrinsic graphics calls that make visual programming fun and easy. Unfortunately, it doesn't conform to any graphics interface standard (GKS, PHIGS, or whatever) and thus lacks high-level routines for axis rotation, scaling, and other coordinate transformations. Still, just about everything you need for graphics power is there.

Data Types

Version 4.0 offers several new data types. There are still six basic types—string, char, Boolean, pointer, integer, and real—but the latter two have more choices.

The *integer* types and their characteristics are shown in Table 1, this page. The fundamental floating-point type in Version 4.0 is the same 6 found in earlier versions. The old Turbo-87 became a has-been with Version 4.0, which offers four additional *real* types compatible with IEEE Standard 754 and usable only on an 80x87 math coprocessor; no IEEE emulation package is available. What's more, the compiler won't let you declare variables of these new types unless you specify the *$N* switch, explicitly stating that your target system has a coprocessor. Table 2, this page, gives the available real types.

Pointers are ordinarily bound to a type, but they can be passed as untyped parameters to a function or procedure. One unfortunate exception to the ANSI standard is that Version 4.0 doesn't support the passing of procedural and functional parameters; that is, procedure *B* can't accept a pointer to function *A*, then pass control to function *A* using the pointer. You couldn't do this in earlier Turbo Pascal versions either, so nothing's changed in that regard, but because the standard mandates it and because it's one of those things that makes C such a powerful systems programming language, Version 4.0 should have supported it.

Here's the Wrap

From the beginning, Turbo Pascal has been controversial on two grounds: its limitations and its disregard of formal standards. With Version 4.0, Borland has removed the limitations. As for standards, you could take the position that with more than 600,000 legal copies available—probably more than all other Pascal compilers combined—Turbo Pascal is the standard by sheer weight of numbers. Frankly, that's what I expected Borland to do inasmuch as success breeds arrogance, of which Philippe Kahn has never been found wanting. Instead, though, Version 4.0 has moved much closer to the formal standard, deviating—though differently in particulars—to approximately the same extent as Microsoft, VAX, and other "purer" Pascal dialects. And that's good for everybody.

I wish Turbo Pascal 4.0 had a debugger and that it exported compiled modules for linking with other languages. But what the heck, it's

Type	Range	Size
byte	0. .255	1
shortint	–128. .127	1
integer	–32768. .32767	2
word	0. .65535	2
longint	–2147483648. .2147493647	4

Table 1: Integer *types and their characteristics*

Type	Range	Digits	Size
real	10**–38. .10**38	11	6
single	10**–38. .10**38	7	4
double	10**–38. .10**38	15	8
extended	10**–4391. .10**4391	19	10
comp	2**63. .2**63–1	8	??
			18.9 digits of precision

Table 2: *Real types and their characteristics*

got everything else, and for $99.95, it's all the compiler that even the most demanding Pascal programmer needs.

by Kent Porter

CodeView

Product:
Microsoft CodeView, Version 2.1
Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later
Price:
Comes with Microsoft MASM 5.0, C 4.0 or later, and FORTRAN
Vendor:
Microsoft, 16011 N.E. 36th Way, P.O. Box 97017, Redmond, WA 98073; (800) 426-9400

I can remember when the hardest thing about debugging code was trying to decide whether to print the whole link map or to just jot down an item or two. Printing the whole thing could take 20 minutes or more. The other tough decision was whether or not to compile the code with listing output so that I could follow my program at the source level.

Thankfully, those days have gone. Symbolic debuggers did away with the need for the link map; source-level debuggers "knew" which source lines went with which machine instructions. But somehow, I still felt there was more a debugger could do for me—such as knowing the type as well as the address of variables, for example; or not intermixing output from the debugger with output from the program under test; or setting it to watch certain variables for me.

To address most of these issues, Microsoft created CodeView 1.0, which came bundled with Microsoft C, Version 4.0. Although memory was still a huge problem, at least the debugger knew as much as I did about the logistics of the program. Version 2.1 relieves the memory problem by making use of expanded (EMS) memory. This is by no means a perfect solution, however.

Microsoft CodeView has two main operating modes. One is purely prompt/command-oriented and is appropriate for use on MS-DOS machines that do not support one of the conventional IBM display adapters. The second is a windowing mode, usable only on MS-DOS machines with compatible display adapters. I'll discuss the windowing mode here, but nearly all CodeView's power is likewise available in command mode.

CodeView is easy to use, but not at the expense of power. Most if not all of Symdeb's commands can be used in a dialog window. This scrolling window contains the customary prompt as well as output from issued commands. It even includes some history so you can go back and look at some output again.

There are several other windows that are much more interesting. First, there's the source window; in it, some portion of your source code, usually the portion being executed, is displayed. The source window has three modes—source only, assembly language only, and mixed source and assembly language. The currently executing line is in a color bar.

The remaining two windows are optional. The register window displays the current value of all the processor's registers (all 32 bits if the processor is an 80386 and you have set 386 mode in CodeView). The watch window displays the current value of arbitrary expressions, which allows you to monitor a variable or a more complicated expression.

You can set breakpoints to any given line simply by tapping F9. The desired line is displayed in high-intensity white, indicating a breakpoint. The dialog version of the breakpoint-set command has more power, allowing the setting of pass counts and commands to be executed upon breaking. F7 executes a given statement.

There are the usual two forms of

(continued from page 126)

single-step—*STEP INTO* (F8) and *STEP OVER* (F10). *STEP INTO* steps into a function call being stepped, whereas *STEP OVER* executes until the stepped function returns. Both modes do exactly the same thing if there is no function call at the current location. Both are sensitive to the current source window mode if, in source-only mode, the single steps are source steps. In the other two modes, steps are single instructions.

One of the most useful additions to any debugger is the ability to use data breakpoints. Data breakpoints allow you to stop program execution when a particular variable, expression, or region of memory changes. Microsoft calls these tracepoints. Of course, you may not be interested in all changes but only in the one change that results in an erroneous value. If you can write an expression that yields zero when all is OK and nonzero when the program should stop, you can use watchpoints.

The main problem with data breakpoints is speed. Most processors don't implement any special mechanism for them, and as a result, a debugger must do a single step, check data breakpoints, single step, check, and so on. It's easy to see why this can be slow. But Microsoft has started to take advantage of the 80386. Tracepoints of limited range can be done while the processor is running at full speed on the 386 because of the processor's debug registers. Note that watchpoints do not take advantage of this feature, and I have no idea why not. To use this feature, you must give */r* on the CodeView command line, as documented in the readme file.

I like mice, so CodeView's mouse support is a big plus for me. Both F7 (go until) and F9 (set breakpoint) are implemented on the mouse buttons, so these operations are simply point and click. You can scroll both the source and dialog windows and you can move the line between them using the mouse.

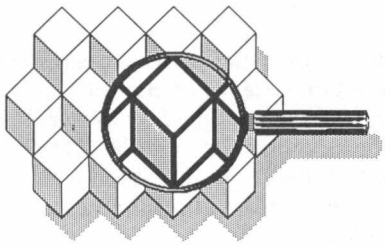
One last feature is the ever-desirable *CONTINUE UNTIL RETURN*. If you have traced into a procedure and then wish to execute until the function returns, simply pull down the Calls menu and point to the function to which you wish to return. The source window then displays that function and the cursor is positioned such that pressing F7 executes the desired result. Note that this feature is available only in window mode, not in dialog mode.

A combination of excellent performance, richness of features, and a large installed base will probably guarantee that CodeView will remain a reference standard in debuggers for some time to come. And despite a few flaws, that's as it should be.

by Richard A. Relph

DDJ

OF INTEREST



Object-Oriented Programming

Productivity Products International is offering an object-oriented software engineering environment that contains Objective-C, Vici, and a catalog of Software-ICs. Objective-C is an object-oriented sofware engineering language that includes such features as encapsulation, which holds data within an object and surrounds the data with a shell of procedures; inheritance, which builds and reuses code; and dynamic binding, which allows Objective-C objects to decide at run time which routine to run. Vici is a prototyping and debugging environment. Software-ICs are modules of reusable code that give programmers the same advantages as integrated circuits give hardware designers.

The PPI product set is available for the DEC VAX, Sun Microsystems, HP-9000, and Apollo lines and the IBM PC AT. Source code licenses are available for users who want to port the environment to their own proprietary hardware. Reader Service No. 16.
Productivity Products International Inc.
27 Glen Rd.
Sandy Hook, CT 06482
(203) 426-1875

A new programming product for the Macintosh is available from **Coral Software**. Object Logo features object-oriented programming tools, an interpreter for interactive debugging, a compiler that automatically translates each procedure into native code, a user interface that conforms to the Macintosh standards, complex and rational arithmetic, arrays, Macintosh ROM support, and a variety of advanced primitives and debugging tools. Object Logo is available for the Macintosh 512, 512E, Plus, and SE and costs $79.95. Reader Service No. 17.
Coral Software
P.O. Box 307
Cambridge, MA 02142
(617) 547-2662

Languages

A Modula-2 language systems is now available from **ana-systems**. Modula-2/68 is suitable for professional software development or academic instruction on systems using the Motorola line of 32-bit microprocessors. With Modula-2/68, program modules can be compiled separately. An executable process can be built by linking with previously compiled program modules or by linking with library procedures written in C. Modula-2 programs can easily access standard C object libraries, which means that modules can use previously written and debugged subroutine libraries written in other programming languages. For special applications based on the MC68000 processor line, the linked Modula-2 programs can be written out in the Motorola S-record format and then easily downloaded to standard development systems.

Modula-2/68 is available for the Convergent Technologies line of Unix systems and sells for $1,200. Reader Service No. 18.
ana-systems
697 Saturn Ct.
P.O. Box 4759
Foster City, CA 94404-0759
(415) 341-1768

WATCOM has announced two new C language products for IBM PC and PS/2 DOS systems. WATCOM C 6.0 is an optimizing compiler that comes with a full range of programming tools, including a windowed source-level debugger. WATCOM Express C provides an integrated development environment, including an editor, compiler, debugger, and run-time library that are all memory-resident. Programs may be compiled in memory and then executed directly without separate link and load steps. Reader Service No. 19.
WATCOM Products Inc.
415 Phillip St.
Waterloo, Ont.
Canada N2L 3X2
(519) 886-3700

C Workshop from **Wordcraft** is a program that teaches C. It also contains a programming environment that users can use for experimenting, prototyping, and developing the modules for any programming project. C Workshop contains a full standard K & R compiler, an editor, a run-time library, tutorial software, and a book. The software uses 220K and runs on 320K MS-DOS computers with industry-standard BIOS. The package sells for $69.95. Reader Service No. 20.
Wordcraft
3827 Penniman Ave.
Oakland, CA 94619
(800) 227-2400

PL/D from **Dair Computer Systems** is a new system language that allows users the speed and control of assembly language with the ease of expression of a compiler. Like assembly language, PL/D has no run-time library overhead. PL/D code is inherently capable of relocation for EPROM-based embedded control, a common PL/D application. Features of the language include macro and conditional compile facilities, library functions that can be incorporated into the program at compile time as SYS source files, 17 options that can be specified within the source, and some features similar to Forth and C. Source code of the compiler is available. PL/D requires a 192K IBM PC AT or PC/XT with MS-DOS 2.0 or later and sells for $124.95. Reader Service No. 21.
Dair Computer Systems
3440 Kenneth Dr.
Palo Alto, CA 94303
(415) 494-7081

Fun Stuff

If you are tired of programming and want to get out of the house, **Ex-**

panded Entertainment has released a new film that compiles some of the top shorts in computer animation in The Computer Animation Show. The show is a collection of 3-D character animation, abstract work, corporate show reels, experimental animation, and several ground-breaking music videos. The show is presented in 35 millimeter and charts the growth of computer graphics over the last five years. The movie debuts in 400 cities nationwide throughout 1988. Reader Service No. 22.
Expanded Entertainment
2222 S. Barrington Ave.
Los Angeles, CA 90064
(213) 473-6701

Hardware

RasterOps Corp. has introduced a high-resolution color graphics board for the Macintosh II. The ColorBoard 1/104 features a single-slot design, true color capabilities, and 1,024×768 pixels on a 24-bit color plane capable of displaying 16.7 million colors simultaneously. The ColorBoard 1/104 is priced at $2,795 and runs on the Macintosh II. Reader Service No. 23.
RasterOps Corp.
10161 Bubb Rd.
Cupertino, CA 95014
(408) 446-4090

The EVERCOM II 24 is a 2,400-bps modem for the IBM PS/2 with Micro Channel architecture. Available from **Everex**, this modem features state-of-the-art signal processing and adaptive equalization, auto-dial, auto-answer, auto-speed-matching to the calling modem, and auto-sense of tone/pulse dialing. Users can easily switch from voice to data and data to voice during a call. Built-in intelligence can detect and respond to such variables as transmission speed and data format. International compatibility is supplied by incorporating Hayes, Bell 212A/103, CCITT V.22, and V.22 bis protocol standards. BitCom, a menu-driven communications program is also included. The EVERCOM II 24 is priced at $299. Reader Service No. 24.
Everex
48431 Milmont Dr.
Fremont, CA 94538
(415) 498-1111

DigiBoard has introduced the OpenEnder, a multichannel board for the PS/2. The OpenEnder features DigiBoard's modular I/O concept, the I/O Mate. The I/O Mate contains I/O components and mounts to the OpenEnder portion of the board that holds the processor and memory. I/O Mates are available in both 4- and 8-port configurations with an added synchronous channel available on the I/O Mate+ series. The OpenEnder features an on-board 80186 microprocessor operating at 12 MHz, 256K of dual-ported RAM, and up to 64K of ROM to store user-defined programs. The OpenEnder is compatible with MS-DOS and Xenix and will support other operating systems. List prices are $1,349 with I/O Mate-8, $1,399 with I/O Mate8+, $1,149 with I/O Mate-4, and $1,199 for I/O Mate4+. Reader Service No. 25.
DigiBoard Inc.
6751 Oxford St.
St. Louis Park, MN 55426
(612) 922-8055

Kinetic Access is a security product from **Kinetic Corp.** that offers micro system security through an easy-to-use program. The total system includes a hardware device that controls the booting process and a resident control program that requires 45K RAM while in operation. The hardware device can be either an EPROM, which fits into an available ROM socket, or a short expansion card that plugs into any available slot on the PC. After installation of Kinetic Access, every time the system is booted up, Kinetic Access password protects users' files and applications. Kinetic Access is available for users of IBM PC, XT, AT, and true compatibles operating with MS-DOS, Versions 2.x or 3.x. The product is priced at $129.95 per unit with EPROM and $149.95 per unit with short expansion board. Reader Service No. 26.
Kinetic Corp.
Distillery Commons 240
Lexington Rd. @ Payne
Louisville, KY 40206-1990
(502) 583-1679

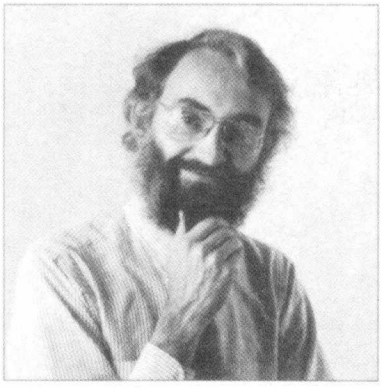
Tools and Utilities

A data-compression package has been released by **Isogon Corp.** NEWSPACE is a RAM-resident utility that significantly increases the storage capacity of any IBM or IBM-compatible PC hard disk. The program automatically and transparently compresses word processing, spreadsheet, database, and all other kinds of data files without any user involvement. Although the kind of compression achieved depends on the nature of the data, 50 percent overall compression is average. NEWSPACE operates with all application programs and RAM-resident utilities and works on all PC-DOS and MS-DOS machines running Version 2.0 or later, including the PS/2. The program sells for $69.95. Reader Service No. 27.
Isogon Corp.
330 Seventh Ave.
New York, NY 10001
(212) 967-2424

Quinn-Curtis has announced Science and Engineering Tools for the Macintosh 512K or bigger. The package includes procedures for general statistics, multiple regression, curve fitting, integration, FFTs, solving differential and simultaneous equations, matrix math, complex math, data smoothing, linear programming, root finding, and special functions. A 160-page manual describes the form, function, and parameters of every procedure and function. All the software tools are supplied on a 3.5-inch disk in both Lightspeed Pascal and Turbo Pascal source code. The Science and Engineering Tools package retails for $74.95. Reader Service No. 28.
Quinn-Curtis
49 Highland Ave.
Needham, MA 02194
(617) 444-7721

DDJ

SWAINE'S FLAMES



I believe that the pace of change in programming methodology is increasing, and that those who learn new paradigms will keep up, while those who do not will be left behind.

I am encouraged in this belief by a rereading of Robert Floyd's acceptance lecture on receiving the 1978 Turing Award. Floyd contended then that "continued advance in programming will require the continued invention, elaboration, and communication of new paradigms."

Floyd took the term *paradigm* from Thomas Kuhn's *The Structure of Scientific Revolutions* and used it to refer to general models of problem solving and the shared conventions and traditions of a discipline. Structured programming is a general paradigm, recursion a narrower one.

In his lecture, Floyd described how he invents new paradigms. Having solved a problem, he next resolves the problem from scratch, then looks for the general rule for solving problems of this sort, ultimately deriving a broad problem-solving paradigm. "Most of the classical algorithms to be found in texts on computer programming can be viewed as instances of broader paradigms," Floyd said. "Simpson's rule is an instance of extrapolation to the limit. Merge sorting is an instance of the divide-and-conquer paradigm. For every such classic algorithm, one can ask, 'How could I have invented this,' and recover what should be an equally classic paradigm."

And that's what you need to do if you want to advance the field and your place within it, according to Floyd: "I believe that the best chance we have to improve the general practice of programming is to attend to our paradigms."

I believe that attending to our paradigms is more imperative today than a decade ago, for two reasons.

First, there are simply more paradigms that we must understand today. Consider this list of vogue topics: logic programming, production systems, expert systems, blackboard systems, functional programming, object-oriented programming, event-driven programming, neural nets, associative memory models, machine learning paradigms, MIMD, SIMD, and data-flow programming. Many of these topics overlap and some may be synonyms, but how many programmers can sort them out? Or predict which will be important two years from now?

Second and ultimately more far-reaching, I believe that a very deep paradigm, the Von Neumann model of sequential processing, is in the process of being supplanted by many parallel-processing paradigms. If this is true, it will be the most fundamental change in programming since the development of high-level languages, and it will radically affect the way we think about the process of writing software. With certain limited and constrained exceptions, all software is written within the Von Neumann paradigm. As programmers we scarcely know how to think in parallel terms. Our algorithms will not transfer. We will need a paradigmatic approach in order to find our way in a parallel world.

The parallel world is bigger, and hairier, than the sequential world.

True, there are grounds for skepticism about parallel processing. There is no architectural platform for parallel programming outside certain specialized areas, such as numerical analysis and graphics. The kind of parallel processing I am talking about—multiple instruction, multiple data (MIMD) programming—is difficult, with significant unsolved problems.

True, there is no parallel equivalent of the IBM PC. But desktop multiprocessor architectures based on the transputer chip exist as commercial products today; they are just not yet cost-effective. The biggest cost factor is the price of the transputer, which is a function of demand and competition. This barrier could start falling within the year. It is not too early to imagine what you could do with a true parallel-processing system.

True, decomposition of a problem into MIMD parallelizable components is hard. That is precisely why a paradigmatic approach is necessary: finding the right paradigm can give you the solution to a broad class of problems. The divide-and-conquer paradigm, for example, is well adapted to MIMD parallelization, so if you can cast your problem in that form, you should be able to find a good parallel solution. And a good parallel solution is one that increases throughput radically.

I believe that the most successful programmers in the next decade will be those who carry in their toolkits, among their shiny metric and nonmetric algorithms, a rich set of paradigms.

Michael Swaine

Michael Swaine
editor-in-chief

#138 APRIL 1988

2.95 (3.95 CANADA)

Dr.Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

CREATING A NEW AI LANGUAGE

COMBINING RULES AND HYPERTEXT LINKS

REVIEWS

- Turbo Professional 4.0
- Guidelines C++
- Brief 2.0

LANGUAGES:

C, Forth, HyperTalk, LISP, Modula-2

0 38351 16562 8 04

ARTICLES

Extending LISP ▶

Creating an Adventurous Language **18**
by Jonathan Amsterdam
Jonathan describes AAL, a language based on LISP that includes features from both object-oriented and declarative languages.

Topics in Knowledge-Based Languages **40**
by Bill and Bev Thompson
The development of expert systems has often emphasized the paradigms of the subject area and their representation, often to the point of hindering rapid development and change. Bill and Bev describe flexible new structures called Topics that give knowledge-based system designers the advantages of both hypertext and declarative languages..

Theorem Proving Using Semantic Resolution **50**
by Anthony J. Dos Reis
For those of you into propositional logic, Anthony offers working code (in C) for experimenting with "mechanical" theorem proving.

REVIEWS

EXAMINING ROOM **136**
coordinated by Ron Copeland
Products examined from the programmer's perspective. This month: Version 2.0 of the Brief editor, Guideline's C++, and the Turbo Professional 4.0 library from Turbo Power.

COLUMNS

C CHEST **98**
by Allen Holub
Improving on an old standard, Allen tears into *printf( )* to add new features.

HyperTalk ▶ **TO THE MACS** **106**
by Stan Krute
Updates on QUED/M, TMON, and HFS Navigator, as well as code for a HyperCard Scouting Toolkit.

Modula-2 ▶ **STRUCTURED PROGRAMMING** **118**
by Kent Porter
Updating Niklaus Wirth with a *LineDrawing* graphics module in (what else?) Modula-2.

Forth ▶ **THE FORTH COLUMN** **130**
by Martin Tracy
A report on the 9th FORML, the latest on the ANSI FORTH effort, and other news.

FORUM

EDITORIAL **6**
by Tyler Sperry

RUNNING LIGHT **8**
by Tyler Sperry

LETTERS **12**
by you

CARTOON **14**
by Joe Sikoryak

SWAINE'S FLAMES **152**
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX **129**
Where to go for more information on products.

OF INTEREST **144**

About the Cover

They said robots were an AI cliche, but we replied, "Hey, we may be April fools, but we're not dummies!" Special thanks to Android Studios, of Oakland, Calif. for central casting, and to Lord Whorfin for the insulating fluid.

Next Issue

May will see a number of changes in *DDJ*, including a new column on "Programming Paradigms." The issue focus is on developing applications from a number of perspectives, as well as the usual surprises.

EDITORIAL

Sizing Up Microsoft

It's been said from time to time that *Dr. Dobb's* has a love/hate relationship with a little software company up in Washington state, a company you may have heard of. Some people who read the magazine often think we're biased in the company's favor because we mention it so often, or because we haven't published a scathing indictment of the bugs in one of their compilers (lately). Other people, who presumably don't talk to the folks in the first camp, charge that we spend too much time covering Microsoftian developments and urge that we spend more time covering (fill in the name of your favorite product category).

While there seems to be almost a tradition of Microsoft "bashing" in the industry, that's not what's going on here. What's really going on is that the company has grown so much in the last few years that we're all becoming victims of shifting paradigms. Although it may be hard for us (especially the old-timers) to appreciate, there's no escaping it: If you're a serious developer in the personal computer marketplace, you have to analyze Microsoft's products and strategies and how that impacts your products and plans. Often that means reading both for what was said, and what wasn't; analyzing both for tactics and strategy.

Which brings us to the point of this little essay: For most developers, Microsoft's role can be seen today as primarily of strategic rather than tactical importance. Increasingly, the details of specific announcements are becoming less important than strategic concerns.

Case in point: OS/2. It took years to develop—big companies move slowly—and it'll take years for the new generation of applications to appear. But despite the tactical opportunity for other operating systems to penetrate the market (in relatively small numbers), the strategic reality is that OS/2 will undoubtably prevail in the coming years unless IBM pulls the plug. OS/2's victory will have little to do with technical merit, and everything to do with the incredible resources of Microsoft and IBM. OS/2 is the Ada of personal computer operating systems: less attractive than other options, but massive and impossible to assail.

In reflecting on this state of affairs you might be disappointed, like many of us, that Microsoft's strategy ignores the 386 market. It is ironic that most of the scheduled acceptance of OS/2 will occur in a year or more, just when (rumors say) we'll all be hip-deep in 386 machines, but this just further emphasizes the strategic choices developers have. You can take the safe route and write applications for OS/2, or consider all those 386 machines crying for new software. A great deal of successful 386 software is being written by small companies, by people who are able to respond quickly to changing market conditions.

And finally, there are times when it's to everyone's benefit to have a big company leading the way. Microsoft's involvement in CD-ROM is a perfect example. The company's extensions for MS-DOS are an important tool for everyone, but the most important concern right now is making CD-ROM viable. In a market deadlocked with the hardware companies are waiting for the software companies and vice versa, Microsoft has taken the laudable step of continually pushing ahead with CD-ROM products and support. Which means we all might get to reap the benefits of CD-ROM before the turn of the century—presumably even those of us running MS-DOS as a task under a 386 version of Unix.

Tyler Sperry
editor

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editor *Rhoda Simmons*
Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Fulfillment Coordinator *Francesca Martin*
Subscription Supervisor *Kathleen Shay*
Newsstand Coordinator *Sarah Frisbie*
Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 129*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, with two special issues per year by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

Subscriptions: $29.97 per 1 year; $56.97 per 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

As my friend Ezra Shapiro has pointed out, when it comes to Artificial Intelligence in personal computer software, we've still got a long way to go. For all our simulations of neural nets and expert systems, our machines still seem woefully inadequate in reasoning power. Today someone told me about an advanced communications package that recognizes when it's been invoked without being configured and actually prompts the user for the information it needs. While I acted suitably impressed, I couldn't help wondering why the program couldn't check the hardware, access a database containing parameters and phone numbers, and furnish the user with a list of assumptions for approval. It could be that I just expect too much; the program my friend described is apparently considered a fairly sophisticated program by his peer group.

Yes, Ezra, still we have a long way to go.

DDJ readers, as a class, probably have less road to travel than other programmers. This issue, after all, has a couple of articles designed to stimulate your interest in expanding what we commonly describe as AI languages. As we investigated articles for this issue, I discovered that for many people the line between AI languages and "ordinary" languages has blurred. For Bill and Bev Thompson, authors of "Topics in Knowledge-Based Languages," (page 40 in this issue), and authors of KnowledgePro, KnowledgeMaker, and MicroExpert, the distinction is no longer relevant to their work—they do most of their expert system *work* in an extended version of Pascal. Their article demonstrates that real world problems often require more flexibility from the language (and by extension the programmer) than you'd expect from the standard AI discussion. After talking it over for hours, Bev said, they'd had to admit that whatever you do, the program will eventually be translated into binary, and they've decided they no longer "believe" in AI languages, per se.

While on the subject of unbelievable things, you'll probably want to check out Edward Yourdon's newsletter, *American Programmer*. The premier issue just arrived and describes the publication as "dedicated to casting a caustic eye on the American software scene." It certainly does that! While Yourdon's perspective is a little more MIS oriented than many of us prefer, the newsletter is filled with perceptive commentary and intelligent discussion of such topics as the economics of the DP industry, and the issues of programmer productivity and costs.

One of the more startling forecasts in the premier issue is what Yourdon admits a rash prediction, the thesis that "the American programmer is about to go the way of the dinosaur and the dodo bird." Yourdon is talking here about programmers in the corporate environment, and he has a substantial body of facts and figures to buttress his thesis that foreign competition is about to do for programmers what it did for the American auto worker a few years back.

Obviously, at a charter subscription rate of (only) $295, this newsletter is not for the casual programmer. But a subscription for the company library would be a sound investment. It's good to be occasionally reminded that there are larger issues than the product announcements, advertisements, and reviews that many industry publications dwell on.

Tyler Sperry

Tyler Sperry
editor

ARCHIVES

Ten Years ago in *DDJ*

"A company which claims it is ready to manufacture and sell truly intelligent household androids, or robots, within the next two years—to the tune of $4,000 apiece—has the nation's artificial intelligence experts up in arms. The experts unanimously state that the claims are fraudulent. The company, Quasar Industries located in Rutherford, NJ, is currently touring shopping centers around the country with a model of an android which they claim comprehends 4,000 words of vocabulary and can vacuum, wash dishes, and teach the kids French. Crowds in the sundry department stores are dazzled, according to numerous press reports. This robot is a fake, according to scientists from Carnegie-Mellon University in Pittsburgh, PA. The scientists recently undertook a first-hand investigation of the matter and found the robot to be a "radio-controlled puppet." Stanford University News Release, "Et Tu Klatu?"—*"Letters,"* DDJ, *February 1978.*

Just another arty fact

"After all, the strange fact about AI has always been that it's easier to simulate an expert than to simulate the general common sense of a five-year-old."—*Michael Doherty,* DDJ, *June 1984.*

It's always so code in here!

"*DDJ* is important as a journal because of the code. It was started to publish code and it has always published code. It publishes more code than any other magazine. And its still my belief that people [programmers] learn from reading other people's code.

What they learn, of course, is how to program well: they pick up tricks, insights, algorithms, all of which are particularly well expressed when presented in a form in which they will ultimately be realized: as code. We hope to go right on publishing useful and educational code, including both programs significant in themselves and bits of code that demonstrate some exemplary algorithm or insight."—*Michael Swaine,* DDJ, *February 1985.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Correcting our Pointers

Dear *DDJ*,

I could not bear to let Clyde Schechter's letter, Optimum Performance?, in the January 1988 issue go unanswered. It will leave young C programmers very confused about both multidimensional arrays and pointers.

Apparently Mr. Schechter has a two-dimensional array confused with an array of pointers. The definition:

```
int ary[10][10]
```

reserves storage for 100 integers, and the storage is quite contiguous. If the storage is not contiguous, then the concepts of pointer arithmetic and scaling in the C language have been built on faulty ground. The variable *ary* is an "array of 10 pointers, each containing 10 integers." The element *ary[0][10]* is the same element as *ary[1][0]*. No pointer variables have been created. The expressions *ary*, *ary[0]*, and so on are all pointer constants. As pointer constants their values cannot be changed, meaning that they cannot appear on the left of an assignment expression. Rows cannot be swapped by swapping pointers, as Mr. Schechter implies. The distinction between pointer constants and pointer variables is fundamental to fully understanding the C language.

I think the definition he describes is:

```
int *ptrary[10]
```

ptrary is "an array of 10 pointers to integers." The storage for the pointers has been reserved, but the pointers have not been initialized. The storage to which they point is not yet present. Dynamic memory allocation can be used to obtain different amounts of storage for 10 different arrays, and each pointer can be set to reference one of the arrays. Now the pointers (array elements) can be exchanged.

Your readers should understand the difference between these next two expressions. *ary[1][2]* references element (1 * 10) + 2, element number 12, (the 13th element counting from zero), of array *ary*. *ptrary[1][2]* references element number 2 (the third element) of the array pointed to by element number 1 (the second element) of array *ptrary*. The scale of each row of *ary* is 10, while the scale of the rows of *ptrary* is unknown.

In support of Mr. Schechter, this topic is one of the most confusing aspects of the language. In *C by Design*, the college level textbook which I am co-authoring, we decided to devote one full chapter to discussing arrays, and another to discussing pointers. It required that much space to explain these subjects. I hope that this letter provided an adequate overview, and thanks for letting me express this opinion.

George Defenbaugh Jr.
Tulsa, OK

Your points are well taken, and you were not alone in your concern. We'd planned to have a reply from the author, Richard Relph, but last minute changes in the January issue prevented that. —Ed.

Poor Richard's Revenge

Dear *DDJ*,

One of the other editors here at *BYTE* pointed out Tyler Sperry's article in your January 1988 issue, "386 v. 030: The Crowded Fast Lane." I was inexplicable drawn to that article—maybe you can guess why.

In his article, Tyler summarized the conclusions I came to in my September 1987 *BYTE* article (in which I benchmarked a number of 80 x 86 and 68xxx machines). To quote his summary: "If you have some experience with benchmarks (or if you read *BYTE* regularly), you can anticipate what he [meaning: me] found: the 80386 outperformed the 68020 in the majority of tests."

I'm not sure how Tyler deduced that, but he's flat wrong. My article said: "Overall, it appears that—and I know I'll catch a lot of flak for this—the 80386 machines outperform the 68020 machines. Of course, the reason for this could well go beyond the *possibility* [my emphasis] that one processor is simply faster than the other." Nowhere have I made the claim that the 80386 processor performs better than the 68020. I'm surprised that Tyler interpreted "the 80386 machines..." as "... the 80386 processor..." My article was an attempt to reveal aspects of performance of systems based on processors, not an attempt to isolate processor performance.

Further down, Tyler states (in reference to the benchmark tests I ran): "...these tests were performed with the intent to test mathematical performance. The only nonmathematical tests were the infamous Sieve and a quicksort routine."

I've got to throw a flag on that one as well; check out two of the other tests I ran: Dhrystone and Fibonacci. Dhrystone is not a purely mathematical benchmark. It contains a mix of operations that involve indirect addressing, comparisons, array operations, and string manipulations. Nor would I consider Fibonacci's only goal to test math operations; it's main objective is to test the instructions involved in recursion.

All in all, Tyler's article was right on. I couldn't agree more with many of his "lessons." I've always admired the work *DDJ* has done, you people put out one of the finest computer magazines around. Keep it coming, and the other *BYTE* editors and I will keep reading it. (Just ask Tyler to read *BYTE* more carefully next time.)

Richard Grehan
BYTE Magazine
Peterborough, NH

Author Tyler Sperry replies:
Regarding your first point, I must apologize. As you point out, even though your article compared 386 and 020 systems you clearly warned your readers that the results could *not* be extended categorically. (Although it could be argued this is exactly what most of your readers wanted.) As the later paragraphs of my piece made clear, both of us agree on this point. Apparently, my left brain took a vacation during the proofing and missed the gaff.

As to the benchmarks, I must confess that characterizing the Dhrystone benchmark as "mathematical" was a mistake. The single aspect of the Dhrystone is the *strcmp* function—it can affect compiler benchmark performance by as much 30 percent—and comparing strings isn't what most of us consider "mathematical." My mistake with the Fibonacci benchmark was a sin of omission; I consider the Fibonacci to be virtually worthless as anything other than a quick-and-dirty test for the way a compiler handles procedure calls, and perhaps I should have spelled this out.

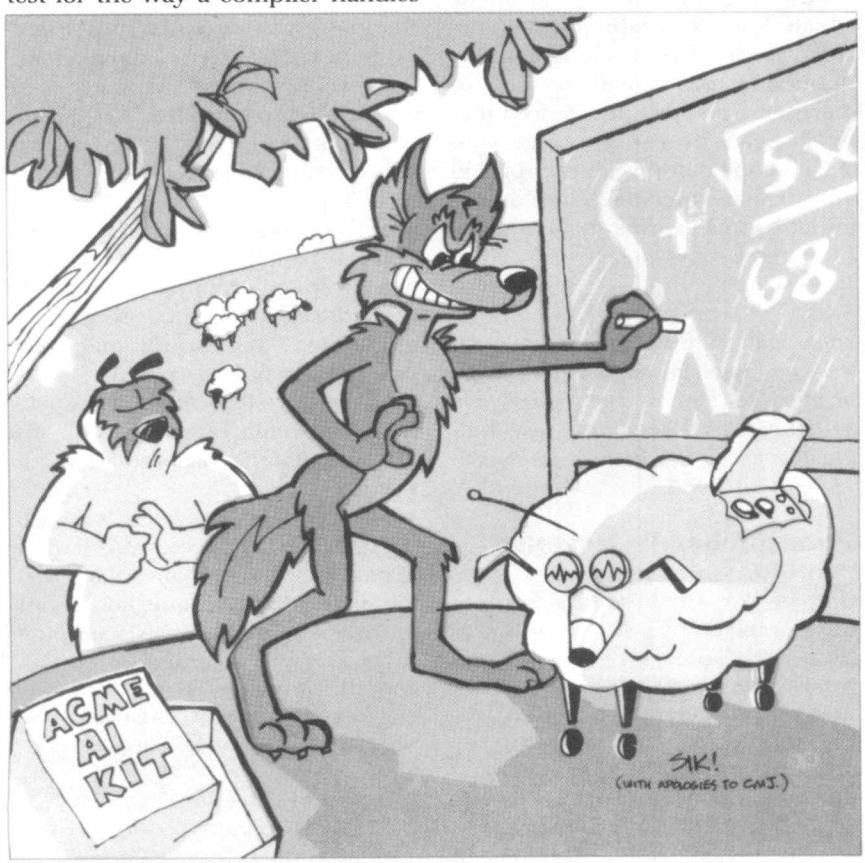
Despite these minor points, I think we agree on the broad issues. It's a pity we have to rely on benchmarks for comparisons of systems—or CPUs.

Dynamic Linking Revisited

Dear *DDJ*,
Congratulations on giving OS/2 some decent coverage in the December issue of *DDJ*.

However, Dave Cortesi's article ("Dynamic Linking in OS/2,") did contain one little flaw. He asserts that DLL s cannot be written in high level languages because of the requirement that SS = DS. In fact, Microsoft C has a switch especially to tell the compiler not to assume DS = SS, and Microsoft uses C to write many of the DLLs that are supplied with OS/2 and Windows.

Ray Duncan
Ex-member of the
Happy *DDJ* Family
Marina del Rey, CA



Early attempts at lambda calculus.

Dear *DDJ*,
I appreciate your article describing the dynamic linking mechanism in OS/2. One minor correction, though. The current C compiler for OS/2 (at least the IBM and the Microsoft offerings) can indeed generate code for dynamic libraries. One must use the *–Au* and *–Gs* options. The *–Au* (usually -Alfu for large model) generates code for SS!=DS. Admittedly not all library modules are available (in particular the I/O library), but one can write shared code in C. With the *–Au* option the compiler generates a save and load for DS for each procedure. *–Gs* suppresses stack checking.

Also, a quick comment on your enthusiasm about the 80386. Yes, it is indeed a wonderful processor. But both OS/2 and the VM systems (Windows/386, Desqview and others) fail to fully exploit the chip. While the VM systems' value comes from compatibility with existing applications (not a minor point), they do not otherwise allow applications to exploit the processor. To fully exploit the native mode (also called protected mode) the path that OS/2 provides has the longterm advantage. Obviously, a merger of the two approaches is needed.

Robert Frankston
(via CompuServe)

Author Dave Cortesi responds:
Duncan and Frankston are correct; both Microsoft C 5.0 and IBM C/2 support the *–Au* option. That generates code at every function's entry to save DS and load it with the selector for the data segment generated for the module, and code to restore the caller's DS on exit. As Frankston notes, the *–Gs* option eliminates the check on stack depth, and that eliminates the commonest source of link-time references to unwanted library procedures.

These options are, however, peculiar to the particular compilers. The IBM Pascal/2 and comparable Microsoft Pascal compilers do not have them, and other C compilers might or might not. And while the features help, they don't fill all the potholes on the road to dynamic linking.

(continued on page 144)

Creating an Adventurous Language

by Jonathan Amsterdam

While browsing around my local computer system not long ago, I had the misfortune to stumble upon the source code for the original Crowther-Woods Adventure program. It was a gut-wrenching display of nonmodularity run rampant. Except for the text of the messages and the network of room interconnections, the entire game, from the properties of rooms and objects to the movements of the bear, troll, pirate and dwarves, was coded right into the program.

I'm sure I wasn't the first to be goaded by this horror into designing a special-purpose language for writing adventure games. But mine differs from others I have seen in its attempt to combine the best of three different programming styles into a single, unified, Adventure Authoring Language, AAL. In this article, I describe AAL (pronounced simply "Al"), emphasizing the roles of the programming styles that comprise it and their implementation challenges.

The programming styles used in AAL all arose from artificial intelligence (AI) research of the 1960s and 70s. Today, each style is typified by a single programming language. The core of AAL involves deductive retrieval from a database of facts and rules, as popularized by the logic programming language PROLOG. AAL borrows the idea of inheritance from object-oriented languages such as Smalltalk for describing the objects of adventure games. And the list-manipulation abilities, syntactic freedom, and general-purpose power of LISP make it the ideal language for implementing AAL.

First, I'll supply an overview of AAL, to give you an idea of how these aspects of the language are realized. Then I'll discuss the programming styles in slightly more detail. Finally, I'll consider some key points of implementation. Because of space limitations, I won't be able to describe all of AAL or its implementation.

Overview of AAL

AAL is designed for writing text adventure games in which players move from location to location and manipulate objects by typing brief commands. AAL maintains the state of a running game in a list of facts called the database. Portions of AAL programs can query the

Jonathan Amsterdam is a graduate student at MIT's artificial intelligence laboratory. He has articles published in several magazines, and may be reached at 617-253-7881.

database to find out about the current state and can add (*assert*) and delete (*retract*) facts to change the state. Facts are represented by lists of symbols; the fact (*in keys house*) could mean that the keys are in the house.

Patterns are used to query the database. A pattern is like a fact but may contain variables, which in AAL begin with an asterisk. A pattern matches a fact if all the constants are identical; the result of a match is to bind the variables in the pattern to the corresponding constants in the fact. So, for example, the pattern (*in *x room*) matches (*in bear room*) with *x bound to bear but does not match (*in bear house*) because *room* and *house* are not equal.

AAL programs cause actions to happen in the adventure game by examining and modifying the database using rules. The rule:

```
((at lamp *x) (on lamp) -> (lit *x))
```

says that if the lamp is at a location *x, and the lamp is on, then *x is lit. Rules consist of zero or more antecedents (patterns to the left of the ->) and one or more actions or consequents. If all the antecedents match the database, then the actions are taken. Variables appearing in the actions have the same values that they were bound to when the antecedents were matched.

Rules can be combined into rule lists, which act like nested *if. . .then. . .else*s. The rule list:

```
(((carrying player rod) -> "The bird is frightened")
  ((not (carrying player cage)) -> "You can't carry it")
  (-> (take player bird)))
```

specifies what happens when the player tries to take the bird in the original Adventure. If the player is carrying the rod, the message "The bird is frightened" is printed and nothing else happens. If the player is not carrying the cage, the message "You can't carry it" is printed. If neither of these conditions hold, then the player takes the bird. (A rule with no antecedents is like an *else*: it always fires. It is also OK to omit the -> entirely in this case.) In general, a rule list can have any number of rules. Their left-hand sides are examined starting from the top, and the first rule to match is activated.

There are several different kinds of actions; you have already seen three. If a string occurs as an action, it is printed. If the first symbol in an action is the name of a built-in AAL procedure, such as *take*, then that procedure is executed (in the case of *take*, the procedure alters the database by asserting the statement [*carrying player bird*] and removing the statement that says in which location the bird is). If an action begins with the word *lisp*, it is given to the LISP interpreter for evaluation; this is an easy action to implement because AAL is written in and runs on top of LISP. If the action can't be classified as one of the above, it is assumed to be a statement to be asserted, as in the case of (*lit *x*).

AAL programs consist of descriptions of the locations, objects, and commands in the adventure game. Listing One, page 58, shows a short AAL program for a simple adventure with only two locations.

The *loc* form is used to describe locations. The first symbol after the word *loc* is the AAL identifier for the location—every entity in an AAL program must have a unique identifier. The next item in the list is a string giving the long description of the location; the identifier, slightly modified, serves as the short description (*the-first-room* is modified to "The First Room").

The remaining elements of the *loc* form are keyword lists—that is, lists beginning with special AAL keywords. A list beginning with the symbol *contains* specifies the items that the room contains at the start of the game. The *exits* list specifies the actions to take when the player attempts to leave the location. Each element of the *exits* list is itself a list, whose first element is the direction the player wants to go (for the first room, west and south are the only choices) and whose remaining elements provide the actions to take.

If an action is a symbol, it is assumed to be the identifier of another location and the player is transferred there without incident. If an action is a string, the string is displayed. So, if the player is in the first room and tries to go west, the list (*w the-second-room*) is examined and results in the player's being moved directly to the second room. If the player goes south from the first room, a message is printed and the player is left where he was. As the north exit from the second room demonstrates, rule lists can be used to implement more complex actions.

The third form in Listing One describes the *blow* command, which the player can enter to blow an object (such as the whistle—see later). The list (*blow *obj*) indicates the syntax of the command: the verb is followed by a single thing, the object of the blowing, which is bound to the global variable **obj*. The *requires* list specifies what conditions must hold for the command to be carried out; this one says that the player must be carrying the object. If not, a message is printed that varies depending on what the object is. This is accomplished using the syntax of Common LISP's *FORMAT* function, a flexible output system similar to but more powerful than C's *printf* function. Because AAL is implemented in and runs on top of LISP, it is very easy to adopt *FORMAT* directly into the language. The last line of the *blow* command description specifies the default action to take; in this case, it is to print a message.

The next form describes the *throw* command. The synonyms *hurl* and *chuck* can be used by the player, but internally only *throw* is used. *Throw*'s syntax is a little more complicated than *blow*'s: you say *throw <some object> at <something>*. The variable **instr* is set to the object thrown and the variable **obj* to the thing at which it is thrown. To throw something, the player must be carrying that thing, and the target must be *here* (at the player's location).

The fifth form in Listing One is an *obj* form describing an object—the game's monster. Each item in an *obj* form is either a feature or a keyword list. In this example the monster has one feature, *fixed*, indicating that it is fixed in place and so can't be taken by the player. Each object in an AAL program may have several features,

which describe various aspects of the object. For instance, a bottle might have the *container* feature, which allows the player to put things inside it; or a door, grate, or chain might have the *lockable* feature, which says that it can be locked and opened with a suitable instrument.

Features also act as predicates; giving the monster the *fixed* feature will result in the fact (*fixed monster*) being asserted at the beginning of the game. In the monster description, there is also one keyword list that describes what to do when the monster is the object of a *throw* command: the monster destroys what is thrown, and a message is displayed.

The next form defines a feature, *treasure*, which describes how to figure the score of an object (using the method of the original Adventure). Because many objects can share the same feature, features provide for great economy of coding. Here you see also that features can take arguments, allowing them to be adapted to different situations.

The next form describes the whistle, indicating that when it is blown, it emits a screech and kills the monster if it's present.

As you can see from the definitions of the *throw* and *blow* commands, the monster, and the whistle, the actions to take on a command are distributed throughout the program. Here's what happens when a command is entered: first, it's parsed according to the template supplied by the verb, and up to three variables are assigned: **command* to the command verb, **obj* to the object, and **instr* to the instrument. The last two are assigned as specified in the syntax template of the command.

Processing the command then begins, in two stages. First, the requirements are checked. The requirements specified with the command itself are checked first, then those on the object, and finally those on the instrument. In my example only the commands themselves have requirements, but it is common for special objects or instruments to place their own constraints on commands.

If all the requirements are satisfied, the command is executed. First, the object is checked to see if it has any actions; if so, they are carried out and processing stops. If the object has none, the instrument is tried, and if it has none, the actions on the command are done. In this way, objects implement their own actions for the most part, and actions specified with commands are the defaults.

This overview of AAL has hit the major features of the language, but unfortunately I have had to omit some features and many details. You will meet a few other aspects of AAL in the rest of the article.

AAL's Programming Styles

Having seen what AAL looks like, let's examine its roots. As I said, AAL combines aspects of three programming styles: logic programming, object-oriented programming, and LISP. Let's start with logic programming.

Logic Programming

In the late 1960s, it was noticed that mechanical theorem provers could be harnessed to answer questions about a database of information.[7] Further refinements led to several powerful AI programming languages, including PLANNER,[2] Conniver,[3] and AMORD.[4] PROLOG is a cleaned-up (but watered-down) successor of these languages. The basic idea behind deductive retrieval is simple: a query—typically a pattern containing variables—is compared against a database of facts, and those facts that match the query are returned.

You can add rules to the picture to make things more interesting (and complicated). There are two kinds of rules: backward rules, which are activated by queries, and forward rules, which are activated when new facts are added to the database. AAL has both these kinds of rules, though the earlier description didn't mention them.

Backward rules have a single consequent and one or more antecedents and are written in AAL with the consequent on the left, like so:

```
((in2 *x *y) <- (in *x *z) (in *z *y))
```

This rule says that something **x* is *in2* something else **y* if **x* is in some third thing **z* and **z* is in **y*. Backward rules are the rules of PROLOG; you could write the preceding rule in standard PROLOG syntax as:

```
in2(X, Y) :- in(X, Z), in(Z, Y).
```

Backward rules are stored in the database along with facts. When a query comes along that matches the consequent of a backward rule, processing continues with the rule's antecedents treated as subqueries. So, if your database consists of the facts:

```
(in water bottle)
(in bottle house)
(in food bag)
(in bag house)
```

and the preceding rule, then the query (*in2 water house*) would match the consequent of the rule, binding **x* to *water* and **y* to *house*; then the query (*in water *z*) is processed and matched against (*in water bottle*) with **z* bound to *bottle*; and finally, the second antecedent of the rule, which is (*in bottle house*) given the current variable bindings, is processed and matched against the identical fact in the database.

Note that the last two facts in the database provide a second answer to the query; if desired, this answer can be found via backtracking. Note also that with rules present, you have to extend the simple pattern-matching algorithm to deal with the case of matching two unbound variables against each other. In this case you simply bind the variables to each other; when either one becomes bound, the other is automatically bound to the same value. This generalized matching process is called unification.

So far, everything I have said is exactly as in PROLOG. Forward rules, however, do not exist in PROLOG (though

they did in PLANNER and related languages). In AAL, a forward rule looks like this:

```
(when-asserted (in *x *y)
  -> (contains *y *x))
```

This rule says that if something **x* is in something else **y*, then **y* contains **x*. More precisely, whenever a fact of the form *(in *x *y)* is added to the database, this rule is activated and the actions on its right-hand side are taken. In this case, the only action is to assert another fact into the database. Backtracking never occurs with forward rules. Forward rules are similar to the production rules of languages such as OPS5.

Note that the rules in Listing One are strictly speaking neither forward nor backward because they are activated by commands, not queries or assertions. But because they behave more like forward rules, they have a similar syntax.

Object-Oriented Programming

Object-oriented programming, typified by languages such as Smalltalk and C++, consists of two major ideas: a program consists of a network of objects that communicate by message passing, and these objects obtain many of their attributes by inheritance.

The second of these ideas is an outgrowth of AI research into frame representation languages,[5,6] and it is the one AAL employs. Objects can have more than one feature, and features may themselves have features, allowing an interconnecting web of inheritance. As in standard object-oriented programming languages, inheritance facilitates the abstraction of common behaviors and properties.

LISP

LISP's enormous power derives from several sources. One source is LISP's ability to model a wide variety of abstractions using higher-order procedures, a feature best exemplified in the Scheme dialect[7] but also available to a large extent in Common LISP. I use this ability to implement *streams*, a data type crucial to AAL's implementation.

Other sources of power are LISP's great syntactic flexibility and list-processing power. Using these, it is possible to implement languages on top of LISP very easily, without the need for bulky and complex lexical analyzers and parsers. Instead, you let LISP's macros and *READ* function take care of that tedium and work at the much more convenient level of lists. And because LISP's interpreter is always present, even when AAL programs are running, you can easily allow AAL programs to escape into LISP, thereby in effect making AAL a superset of LISP.

Implementing AAL

There are numerous interesting aspects of AAL's implementation, but here I only have space to discuss a few of them. The most difficult and interesting part is the deductive retriever, or deducer as it's called in AAL, so I'll consider that first.

Listing Two, page 58, shows the entire code for the deducer. The interface to the rest of AAL consists of the four functions *assert*, *retract*, *deduce*, and *deduce-pattern*. *Assert* is responsible for adding a fact or rule to the database. It only adds something if it is not already present. If it does add a fact, assert checks the left-hand sides of the forward rules to see if any apply and executes the actions of all those that do apply.

Retract removes a fact or rule from the database if it is there. AAL also allows rules to trigger when facts are retracted, and *retract* handles this.

Deduce and *deduce-pattern* are far more complicated, so I'll look at them in great detail. *Deduce* takes two arguments—a list of patterns that make up the query and a list of variable bindings. *Deduce* augments the list of bindings it is given initially with bindings for variables in the pattern list and returns a stream of augmented variable bindings. (I will explain streams later; for now you can think of them as lists.) From these augmented bindings, the caller of *deduce* can obtain values for the variables in the query. The pattern list is taken as a conjunction, so if called with the list *((in *x *y) (in *y *z))*, *deduce* will return a stream of all bindings for **x*, **y*, and **z* that satisfy both patterns.

Deduce begins by checking to see if the list of patterns is empty; if so, the query is trivially true and *deduce* returns a single-element stream consisting of the current bindings. Again, you can think of stream operations such as *stream-cons* as their list equivalents, and you can treat **empty-stream** as nil.

If the list of patterns is not empty, *deduce* first obtains a stream of bindings statisifying the first pattern, using the *deduce-pattern* function. Then it calls itself recursively on the remaining patterns for each binding in that stream, appending all the resulting binding streams into one large stream (that is the function of *stream-mapcan*).

Deduce-pattern simply calls *deduce-pat* with the pattern, bindings, and a list of facts and rules from the database that might possibly unify with the pattern. In its simplest form, *find-possible-unifiers* could just return the entire database all the time; I will make use of a slightly more sophisticated indexing scheme that can cut down significantly on the number of facts and rules returned.

Deduce-pat takes a pattern, a list of bindings, and a list of possible unifiers and returns a stream of augmented binding lists. It first checks to see if there are any more possibilities. If not, it returns the empty stream, indicating failure. Otherwise, it takes the first possibility and tries to unify it with the pattern, renaming its variables first if necessary.

If the unification fails, *deduce-pat* calls itself recursively on the remaining possibilities. If the unification succeeds, deduce-pat calls deduce on the antecedents of the rule (which will be null if the rule is actually a fact) and the bindings produced by the unification. It appends the resulting stream to the stream obtained from calling itself recursively on the remaining possibilities and returns the result. This call *to stream-append*, like the earlier call to *stream-mapcan*, is necessary to ensure

that every possible binding of the query variables is found.

Several internal functions of the deducer deserve mention. The *unify* function takes two patterns and a set of bindings and tries to unify the patterns. It returns the bindings augmented with new ones if it can unify and the symbol *:fail* if it can't. It proceeds by recursively *cdr*ing down the patterns, trying to match each element. Variables are handled by *unify-var*: unbound variables are bound and the match succeeds; bound variables are treated like their values, by calling *unify-const*. *Unify-const* just compares the two elements for equality, returning the existing bindings if they are equal and *:fail* if not. This unifier does not handle patterns containing nested structures as does PROLOG's, but it could easily be modified to do so.

Variable bindings are implemented as LISP association lists (alists). The *add-binding* function just *cons*es a new pair onto the binding list, and the *var-value* function uses LISP's built-in *assoc* function repeatedly to find the value at the end of the chain of bound variables.

The functions *add-to-database*, *remove-from-database*, and *find-possible-unifiers* implement the database indexing scheme. This scheme is quite simple: facts are stored by their first element, on the element's property list. For instance, all the facts beginning with *in* are stored together. Backward rules (the only kind of rule stored in the database) are stored by the first element of their consequent, unless that element is a variable, in which case the rule is stored under the * symbol. The possible unifiers for a pattern whose first symbol is a constant include those on the property list of the constant and the rules stored under *. If the first element of a pattern is a variable, you are forced to search the whole database.

This indexing scheme has the advantage of generality. Further improvements could be obtained, still preserving generality, by indexing off more than the first element; see note 8, Chapter 14, for one such technique. Because many relations, such as *in*, will be ubiquitous in adventure games, they can be treated specially to improve efficiency even more. For instance, if the program maintains, with each object, a list of things that that object contains, then finding the values for *x in the pattern (*in *x object*) is just a matter of fetching that list. Many such optimizations are possible, but keep in mind that they are efficiency hacks, to be hidden in the database indexing mechanism and not made a feature of the language. The view of the adventure world as a single database of facts is a great strength of AAL.

Streams

If streams are just lists, then *deduce* will find every possible binding of the variables and return a list of all of them. This could be a big waste of time if you want only the first binding. And if (as is possible) there are an infinite number of answers, then *deduce* will never return. It would be nice if *deduce* worked more like PROLOG, returning the first answer as soon as it is found but remembering where it was so that subsequent answers could be requested. In fact, this is just how *deduce* works because streams are not lists but special data structures with a rather interesting property.

Streams behave much like lists—they have a *car*, which is a value, and a *cdr*, which is another stream (possibly the empty stream). They are constructed using an operation such as *cons*. The crucial difference is that a stream's elements are not all evaluated; only the first is. The evaluation of the remaining elements is delayed until they are actually requested.

Compare the two expressions:

```
(cons 1 (cons 2 nil))
```

and:

```
(stream-cons 1 (stream-cons 2 *empty-stream*))
```

In the first of these, the inner call to *cons* is evaluated, then the outer one, resulting in the list *(1 2)*. But in the second expression, the inner call to *stream-cons* is not evaluated immediately. Its evaluation is delayed, and the resulting object looks something like:

```
(1 [delayed computation of (stream-cons 2 *empty-stream*)])
```

(This notation is merely for explanation; you'll see the actual implementation in a moment.) When *stream-car* is called on this object, *1* is returned immediately. But calling *stream-cdr* forces the evaluation of the delayed computation. This results in a new stream, which you can write as:

```
(2 [delayed computation of *empty-stream*])
```

Another call to *stream-car* will produce *2*, as it should, and calling *stream-cdr* again will result in the empty stream. So, using *stream-car* and *stream-cdr* on a stream is just like using *car* and *cdr* on a list, except for when the work is done.

The implementation of streams is remarkably simple, provided you have a lexically scoped LISP such as Scheme or Common LISP (see Listing Three, page 62).

You can implement them in terms of two primitives—*delay* and *force*. *Delay* takes a LISP expression and turns it into a delayed computation; *force* takes a delayed computation and evaluates it.

To delay a computation, all you have to do is enclose it in a function of no arguments, so an expression such as *(+ 2 3)* becomes *#'(lambda () (+ 2 3))* (in Scheme, the # and ' characters aren't necessary). You can write *delay* as a simple macro:

```
(defmacro delay (thing)
  `#'(lambda () ,thing))
```

To force a delayed computation, you need only funcall it, so you can write *force* like so:

```
(defun force (thing)
  (funcall thing))
```

You can implement a stream as a LISP dotted pair (*cons* cell) whose *car* contains the *car* of the stream and whose *cdr* contains the delayed computation. Constructing a stream from a value *v* and a computation *c* can be done by writing *(cons v (delay c))*, which is just what *stream-cons* does:

```
(defmacro stream-cons (thing comp)
  '(cons ,thing (delay ,comp)))
```

Note that *stream-cons* must be a macro so that the second argument is not evaluated.

The other stream functions are simple. *Stream-car* is identical to *car*, whereas *stream-cdr* has to force the *cdr* of the stream object. You can represent the empty stream with nil, making the *stream-empty?* predicate equivalent to null. With these functions in place, you can write more complex ones such as *stream-mapcan* and *stream-append* just as you'd write their equivalent list-manipulating versions. The remaining trick is to delay the second argument of *stream-append* so that only the first argument is evaluated.

What is the effect of using streams in *deduce*? Instead of computing all the answers before returning any, *deduce* will take the first answer it finds and return a stream whose *car* is that answer and whose *cdr* represents the rest of *deduce*'s work. If only the first answer is desired, the rest of the stream can be thrown away and no more work will be done. If another answer is wanted, calling *stream-cdr* on the stream will generate it. In this way, only as many answers as required are actually computed.

Streams have many useful applications besides this one. For an excellent description of streams and their uses, including a PROLOG-like interpreter similar to the one I've presented here, see note 7.

Applications of the Deducer

Let's consider three uses of the deducer in the implementation of AAL: rules, the *every* action, and requirements checking.

Much of the work in a typical AAL program is done by rules. A typical rule might say:

```
((at lamp *x) (on lamp) -> (lit *x))
```

To evaluate the rule, you first see if the antecedents (the patterns to the left of the -> symbol) can be satisfied, and if so you execute the actions on the right of the -> with the bindings for the variables on the left. Only the first set of bindings that matches the antecedents is used, so you really do not need the full power of streams here. The implementation is simple:

```
(let ((bindings-stream (deduce (rule-antecedents rule)
                               bindings)))
  (if (stream-empty? bindings-stream)
      :did-not-fire
      (do-rule-actions (rule-consequents rule)
                       (stream-car bindings-stream))))
```

Here, bindings are whatever bindings are in force at the time. If the rule occurs at top level, these will be the bindings of the AAL global variables. But rules may also occur as actions in other rules, so the bindings may contain variables accumulated from those rules.

As another application of the deducer, consider a kind of AAL action I haven't yet mentioned—the *every* action. It allows an action to be taken for every possible binding of a variable. For instance, to "take inventory"—that is, display the items that the player is carrying—you could write:

```
(every *x (carrying player *x) -> (lisp (print *x)))
```

Implementing the *every* action is trickier than it might seem. The basic idea is to obtain the stream of bindings from using *deduce* on the antecedents, then execute the consequents for each binding. The problem with this approach is that a binding may be repeated if the deducer can derive it via different routes. The solution is to turn the stream of bindings into a list, then remove the duplicates. The implementation is shown in Listing Four, page 62.

As my final and most complex example, consider how the requirements for an action are checked. The compiler parses each *requires* specification into a list of requirement structures, each of which has three fields: a pattern to be checked against the database; a string to output if the pattern fails; and a Boolean variable called *succeeded?*, whose use I'll describe later. These structures (along with everything else mentioned when an AAL entity is defined) are stored on the property list of the entity's identifier.

Requirement checking is done by the *satisfies-requirements* function, which is called with the command, object, and instrument of the user's command and calls *check-requirements* for each of the three. *Check-requirements* gets the list of requirements for the command from the given entity, sets their *succeeded?* fields to nil, then calls *check-reqs* with the requirements list and the list of bindings of AAL's global variables. If *check-reqs* returns *t*, so does *check-requirements*; otherwise, *check-requirements* displays the string returned by *check-reqs* and returns nil.

Check-reqs' job is to return *t* if all the requirements can be satisfied or the string that should be printed if they can't all be satisfied. If AAL didn't allow different strings to be associated with each pattern, then you could represent the requirements as a list of patterns and implement *check-reqs* very easily using *deduce*:

```
(defun check-reqs (reqs bindings)
  (if (stream-empty? (deduce reqs bindings))
      "The requirements could not be satisfied"
      t))
```

But the presence of strings for each pattern doesn't

(continued from page 28)

permit this approach because, when *deduce* returns an empty stream, you don't know which pattern was responsible for the failure. You need to use *deduce-pattern* on each pattern individually. One plausible first attempt might look like Example 1, below. Unfortunately, this isn't correct. Because of backtracking, it is a little tricky to pin down just which requirement is to blame when failure occurs.

Consider the requirements of lighting a lamp, which stipulate that the lamp contain batteries that aren't dead:

```
(requires ((in *x lamp) "Nothing is in the lamp")
          ((battery *x) "There are no batteries in the
                                                    lamp")
          ((not (dead *x)) "The batteries are dead"))
```

You would like a message to print out only when the corresponding pattern is the one responsible for failure. But the guilty pattern is not always the last pattern that fails.

Consider a situation in which there are two things in the lamp, only one of which is a battery, and the battery is dead. Say that when *deduce-pattern* is called with the first pattern *(in *x lamp)*, it returns a stream whose first element is a binding list with **x* bound to the battery. Then the second pattern (*battery *x*) succeeds, but the third one fails because the battery is dead. A message should not be printed out yet because there might be other, nondead batteries in the lamp. So you eventually backtrack to the first pattern and get the second binding for **x*, which is the other object in the lamp. Now the second pattern fails right away because the second object is not a battery. Clearly, you would like to say that the reason why the lamp could not be lit was that the batteries in it were dead; the fault is with the third pattern, not the second. But the version of *check-reqs* in Example 1 will return "There are no batteries in the lamp."

The key to the right solution is to notice that once a pattern has succeeded once, it cannot be the cause for

```
(defun check-reqs (reqs bindings)
  (if (null reqs) ;; all requirements passed--success
   T
   ;; else, check the first
   (let* ((req (car reqs))
      (binding-stream (deduce-pattern (requirement-pattern req)
                      bindings))
      (result))
    ;; if it failed, return its failure string
    (cond
     ((empty-stream? binding-stream)
     (requirement-failure-string req))
     ;; else, try all the bindings on the other reqs until
                                                    success.
     ;; dostream just iterates over the elements of the stream.
     (t
     (dostream (binds binding-stream)
      (setq result (check-reqs (cdr reqs) binds))
      (if (eq result t)
        ;; the remaining requirements passed--success
        (return-from check-reqs t)))
     ;; No other bindings worked; return the last failure string
     result)))))
```

Example 1: *An incorrect attempt to use deduce-pattern on each pattern individually*

ultimate failure, even if it later fails. Only a pattern that never succeeds can be guilty. The correct version of *check-reqs* uses the *succeeded?* field of the requirement structure to record this fact; it is given in Listing Five, page 63.

Review

You've seen how AAL combines three programming styles into a powerful language. At AAL's heart is the deductive-retrieval or logic-programming style of PROLOG; AAL represents the state of an adventure game as a database of facts, and AAL programs work by examining and manipulating this database. The pattern-matching, rule-following, and backtracking abilities of the deducer make it easy to write powerful rules and conditions simply.

AAL uses the idea of inheritance from object-oriented languages to ease the definition of game objects, locations, and commands. Each of these entities can possess one or many features, which not only act as abbreviations for properties but also serve as predicates in the database. Features can take arguments and may themselves have features, allowing complex networks to be constructed. Using features to group commands, locations, and objects can also result in considerable economy in rule writing because a whole group of entities (for example, commands that move the player) can be described with a single pattern.

Finally, AAL would have been an order of magnitude more difficult to implement were it not written in and on top of LISP. LISP's reader and macro facility trivialized the parser and compiler for AAL. LISP's property lists, built-in symbol table, and support for association lists simplified many aspects of the implementation. Its ability to construct and call functions on the fly made possible the implementation of the vital *stream* data type. And by allowing AAL programs to escape to the underlying LISP system, I obtained hundreds of useful functions for free.

What Next?

Here I have offered only a sketch of AAL, which is itself only a small part of what could be done in this direction. Many of the fine points of AAL and its implementation can be gleaned from reading the source code. But the deducer stands on its own as a useful program, or it can serve as the beginning of a PROLOG implementation. I have taken care to have all the essential code for the deducer published with this article, so you can begin without delay.

I realize that Common LISP is, despite its name, not terribly common compared to languages such as C and Pascal. But other LISPs, such as XLISP and Scheme, are available and will do just as well. And it is certainly possible to translate AAL into a language such as C, though I would guess the code size would more than double. The major difficulty concerns streams, which cannot be implemented in their full generality in a language that does not allow run-time creation of func-

tions. Ersatz streams designed expressly for their role in the deducer can be implemented, however, by allocating a structure (or record) and storing the relevant local variables in it directly.

I have hardly mentioned the natural-language aspects of AAL because they are secondary to the concerns of the article and are not well developed in any case. Much could be done with existing natural-language technology to improve the parser and the treatment of command execution. I would guess that the conceptual-dependency representation of Roger Schank and his students might prove useful in this domain (see, for example, note 9).

Let me close with a specific, rather ambitious suggestion for improving the natural-language part of AAL. One weakness of the language is that it forces commands to be specified fully: you must say "throw axe at dwarf" instead of just "throw axe," as the original Adventure would let you do, or "blow whistle" instead of just "blow" when the whistle is the only blowable thing you're carrying. Adventure's resolution of the ambiguity was extremely ad hoc.

One reasonable solution when something is omitted from a command would be to look around for a thing that satisfied the requirements of the command and, if there was only one such thing, to use it. That would take care of blowing the whistle, but not killing the dwarf, and not a case in which the player said "drink" and was carrying a bottle of water, but the cap was on the bottle and the bottle was in a locked chest.

A more general mechanism would determine the player's desired action (drink the water in the bottle), construct a plan to achieve it (open the bottle, open the chest—do I have the key?), and execute the plan. It would be a very ambitious programmer indeed who tried to implement this idea; a good solution would probably be worth a Ph.D. thesis. In fact, see Allen's[10] for a start.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Notes

1. Bertram Raphael, "SIR: A Computer Program for Semantic Information Retrieval," Semantic Information Processing, ed. Marvin Minsky (Cambridge, Mass.: MIT Press, 1968): 33-145.
2. Carl Hewitt, "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot," *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI)* (1969): 295–301.
3. Gerald Sussman and Drew McDermott, "Why Conniving Is Better Than Planning," MIT AI Lab Memo, no. 255A (1972).
4. Johan de Kleer; Jon Doyle; Guy Steele, Jr.; and Gerald Sussman, "AMORD: Explicit Control of Reasoning," : 345–356.
5. Marvin Minsky, "A Framework for Representing Knowledge," *Readings in Knowledge Representation*, eds. Ronald Brachman and Hector Levesque (Los Altos, Calif.: Kaufmann, 1985): 245–262.
6. R. Roberts and Ira Goldstein, "The FRL Primer," *MIT AI Lab Memo*, no. 408 (1977).
7. Harold Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs* (Cambridge, Mass.: MIT Press, 1985).
8. Eugene Charniak, Christopher Riesbeck, and Drew McDermott, *Artifical Intelligence Programming* (Hillsdale, N.J.: Lawrence Erlbaum, 1980).
9. Roger Schank and Christopher Riesbeck, *Inside Computer Understanding* (Hillsdale, N.J.: Lawrence Erlbaum, 1981).
10. James Allen, "Recognizing Intentions from Natural Language Utterances," *Computational Models of Discourse*, eds. Michael Brady and Robert Berwick (Cambridge, Mass.: MIT Press, 1983): 107–166.

DDJ

(Listings begin on page 58.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Topics in Knowledge-Based Languages

Presenting a workable compromise between the rigid structures of Prolog, and the chaos of hypertext.

by Bill and Bev Thompson

If you look at the conference proceedings of the American Association for Artificial Intelligence, you'll see the papers broken down into categories such as AI architectures, cognitive modeling, knowledge representation, machine learning, and knowledge acquisition. The key words here are *architecture*, *modeling*, *representation*, and *machine learning*. The interest of much AI work has been to mimic human intelligence through the design of structures that let machines store and manipulate knowledge.

The users of AI systems, on the other hand, are interested in a totally different aspect of the problem. The lure of AI to corporations, scientists, educators, and individual authors is in the ability to communicate expertise via the medium of the computer. The current interest in hypertext systems represents another approach to the use of the computer to communicate complex knowledge. Because the ability to build structures is necessary in order to turn information and data into knowledge, any scheme that hopes to manipulate knowledge must be built around powerful but flexible structural concepts.

Bev and Bill Thompson are directors at Knowledge Garden Inc., Nassau N.Y. They have written extensively about AI systems and are the authors of KnowledgePro, KnowledgeMaker, and MicroExpert.

When we began our design project, which eventually became the language KnowledgePro, our goal was to build a system in which the focus was on the communication of expertise as opposed to its representation. This may sound like a meaningless distinction, but the switch in perspective affects the complexion of the system. In any design process, the designers are forced to make compromises. In saying that we would emphasize communication, we meant that when we made design decisions, we would compromise on the side of ease of use and expressive power rather than code efficiency.

In this article we will describe a structure called a *topic* that is the basic structural and storage unit used in KnowledgePro. A topic is a simple, uniform structure that provides both data representation and control. Topics serve many purposes; they can behave as procedures, functions, variables, system commands, and hypertext nodes. We'll discuss how topics support backward chaining, inheritance, and list processing as well.

Topics As Procedures

In its most basic incarnation, a topic is very much like a procedure in a language such as Pascal. It begins with a declaration of its name and parameters and ends with an *end* statement. Between the beginning and the end, a series of program statements are included. Example 1, page 41, shows an example of several nested topics.

Before discussing the topics in this example, we should say a few words about KnowledgePro syntax. KnowledgePro statements consist of the name of the command followed by a set of parentheses that enclose any parameters to be passed to the command. Each command ends with a period. Several of the most commonly used commands also have shorthand notations that make them more natural to express. Strings that include spaces or delimiting characters are enclosed within single quotes. Words included between (* and *) are comments.

KnowledgePro programs are called knowledge bases. All commands in the knowledge base are associated with an undeclared topic called *!main*. Topics nested within *!main* are executed in one of two ways: directly using a *do* command or through a mechanism called backward chaining. Using the *do* command is just like executing a procedure in C or Pascal. The command can be written using the uniform notation that uses the name of the command:

```
do ('which language').
```

or with a format more familiar to C and Pascal programmers:

```
'which language'( ).
```

In either case parameters can be passed to the topic much as in C or Pascal.

Backward Chaining

Backward chaining is a technique used in PROLOG and by rule-based expert system shells. In a "conventional" programming language, when you use a variable that hasn't yet been assigned a value, the value of the variable is unpredictable. In a language that uses topics, there are no variables as such. The topic serves not only as a structural unit but also as the unit of storage. Values can be both assigned to and retrieved from a topic. The dual nature of the topic makes it possible for an unassigned value to cause the execution of a topic in an effort to find its value.

As an example, suppose we change the first line of the knowledge base in Example 1 to read:

```
if ?'more about languages' is yes
   then do ('which language').
```

and we add the following topic:

```
topic 'more about languages'.
   ask ('Do you want to know more
about languages ?',
        'more about languages',
                                  [Yes,No]).
end. (* more about languages *)
```

Here, the *?* is a shorthand notation for the *value_of* command, so this expression can also be written as *value_of ('more about languages')*. *value_of* requests a value for the topic *more about language* and, not finding a value, causes the commands associated with the topic *more about languages* to be executed. This topic contains an ask command that uses three parameters: the question, the name of the topic in which the answer is saved, and optionally a list that will place a menu of options on the screen. In this example the answer to the question is saved in a topic that already exists. If the topic selected was one that was not defined in the knowledge base, it would be created.

If a statement containing *?'more about languages'* is encountered again in the knowledge base, it will not trigger the execution of the *ask* command but will merely retrieve the value of the topic. Of course, a topic can be considerably more complex than the one shown in the example. A topic whose execution has been triggered by the *?* operator could trigger the execution of other topics either directly or through backward chaining.

Topics As Variables

The *?* operator retrieves the value from a topic. A topic may be assigned a value either as the result of executing commands nested within itself, or it may be assigned a value as the result of another topic's actions.

Along with values and associated procedures, topics have a series of properties that describe the number and kinds of values they may take on. By default, topics are unrestricted in the number and kinds of values that may be assigned to them. It is possible, however, to restrict topics to a range of numerical values, to a limited number of values,

```
topic 'which language'
  ask ('What #mlanguage#m do you want to know more about ?',want,
     [Pascal,C,Lisp,Prolog]).
  do (?want).

  topic language.
    window ().
    say('
         Some people make a distinction between AI and
         "conventional" programming languages.  Though individual
         languages certainly differ, for the most part, an AI language
         is one that was developed at a place where they happen to be
         doing AI.').
    close_window ().
  end. (* language *)

  topic Pascal.
    say ('
        When using Pascal to solve "AI types" of problems, you
        usually have to design low-level routines to implement linked
        list structures.').
  end. (* Pascal *)

  (* topics for C, Lisp and Prolog would go here *)

end. (* which language *)
```

Example 1: *Fragment of a KnowledgePro program, or knowledge base*

```
say ('One of the powerful features of #mLisp#m is the ability
to easily manipulate #mlist structures#m.').

topic mark (find).
  text = read ('threads.fil', concat ('/',?find), '/end').
  window ().
  say (?text).
  close_window ().
end. (* mark *)
```

Example 2: *Using the default topic* mark *with hypertext*

```
topic animal.
  :legs = 4. (* The default number of legs for an animal *)
  dog ().    (* These commands are used for initialization *)
  cat ().
  bird ().
  say ('
       A dog has ',?dog:legs,' legs.
       A cat has ',?cat:legs,' legs.
       A bird has ',?bird:legs,' legs.').

  topic dog.
  end. (* dog *)

  topic cat.
  end. (* cat *)

  topic bird.
    :legs = 2. (* override the default *)
  end. (* bird *)

end. (* animal *)
```

Example 3: *Nested topics showing how values are inherited*

(continued from page 41)

or to members of a set of specified legal values.

Topics may be predefined in the knowledge base or created on the fly. This has the advantage of allowing the creation of complex structures at run time. For example, the statements:

```
ask('Which cat do you wish to choose
?',cat_name, [Figaro,Flo, Babe]).

?cat_name = 'needs a new flea
                          collar.'
```

will produce a menu of choices and allow the user to choose one or several names. For each name chosen, a topic will be created if it doesn't exist and the topic will be assigned a string.

Designers of structured languages tell us that all data structures should be predefined and strongly typed. A reason often cited for this is that programs designed in this way are easier to debug and maintain. Actually, debugging and maintenance are functions of the programming environment. Strong typing and predefinition of data structures make the design of efficient compilers easier because at run time the program contains the location and type of most of its data structures. Languages that create structures at run time are dependent on efficient search routines to find structures.

Rules

In the jargon of expert systems, an *if. . .then* statement is called a rule. Rules can be expressed in the form shown earlier but can also share the formal notation of all KnowledgePro commands. The rule in our example has the following internal representation:

```
rule(eq(?('more about languages'),
yes), delay([do ('which language')])).
```

delay is used to prevent the second half of the rule from evaluating until the first part is evaluated and returns a Boolean value of *T*.

As you might suspect from looking at the example, system commands are themselves topics defined internally but behave in all other ways like user-defined topics. In an application the internal topics can be overridden by defining a topic with the same name within the knowledge base or by using a topic written in another language and included in an external library. The same technique can be used to create new topics that extend the language.

Hypertext and Topics

Returning to the example in Example 1, let's look at the first line in the topic *which language*:

```
ask ('Which #mlanguage#m would
you like to know more about ?',
  want, [Pascal,C,Lisp,Prolog]).
```

We've already described how the *ask* command works, but if you look closely, you'll notice that the text of the question includes the word *language* enclosed within the characters *#m*. This notation is used to define a hypertext node. Any text marked in this way is displayed on the screen in inverse video (or in any color selected by the application designer). The user of the application can use a function key or a mouse to move among highlighted concepts. When a concept is selected, the topic of the same name—for example, *language*—is executed. The commands, *window*, *close_window*, and *say* are self-explanatory. Here, *window* and *close_window* use default values, but they can also include parameters that specify title, color, size, and location of the windows selected.

If no topic with the same name can be found, the system will next search for a topic called *mark* and, if it finds it, will pass the hypertext phrase to *mark* as a parameter. Example 2, page 41, shows an example of one way in which this feature is used. In the knowledge base in Example 2, all the text threaded to hypertext nodes is stored in an external data file called *thread.fil*. When the *say* command displays its message, the *Lisp* and *list structures* are highlighted, but no topics with these names are defined. If the user selects one of these—for example, *Lisp*—the topic mark is executed and passed *Lisp* as a parameter. The first line in *mark* assigns a value to the topic *text*, which is also created at this time. The value assigned is the result of the *read* command.

read uses three parameters: the name of the file to read, where to begin reading, and where to stop reading. The command *concat* returns */Lisp*, which is the result of concatenating the character / onto the value of the topic *find*. All text following this string and up to the string */end* is assigned to the topic *text*. A window is opened; the value of *text*, which might be several pages long, is displayed; the window is closed; and control returns to the statement from which the hypertext was selected.

Nested Topics and Inheritance

Topics can be nested inside other topics to form a hierarchy, which allows you to package pieces of a knowledge base into self-contained units. A topic defined within another topic is normally accessed only from within that topic. Normally, KnowledgePro searches for a topic by examining the topics defined within the current topic. If none of these is the one being sought, it examines the parent topic to see if the topic is defined there. KnowledgePro doesn't look at topics nested within those defined at the parent level. Nested topics are normally invisible from the outside; they are only found through the hierarchical search process. The search continues up the hierarchy until the topic is found or the topic *!main* is examined.

Example 3, page 41, shows an example of a nested topic. If *dog* is referenced outside of *animal*, it won't be found unless the full name *animal:dog*, which defines its location in the hierarchy, is specified. The hierarchical structure allows topics to inherit values from other topics. Example 3 shows a knowledge base that describes some properties of some common animals. The com-

mand *animal ( ).* will display the following message:

```
A dog has 4 legs.
A cat has 4 legs.
A bird has 2 legs.
```

The default value for *legs* is set to *4*. The topics *dog*, *cat*, and *bird* are then executed. Of these topics only *bird* contains an override value. The *:* before the name *legs* in this topic tells KnowledgePro that this *legs* is a local topic, not the one defined in *animal*. Because this topic doesn't exist, it is created. When a value is sought for each of these items in the *say* statement, both *dog:legs* and *cat:legs* are assigned the default value of *4* because no override was specified.

List Processing

Lists are integral to all topics. Each parameter passed to a topic is actually a list. For example, the first parameter of an *ask* command is the question. So far, you've seen questions made of simple text expressions; however, because each parameter can be a list, more complex displays can be created, as in this example:

```
ask (['The total cost is $',
?price * ?'number of items',
   'How will you pay ?'],
            pay,[Check,'Credit card']).
```

Here, the question is a list containing three items: a literal string, the result of multiplying the value of the topic *price* by the value of the topic *number of items*, and a final literal string.

Turning back for a moment to Example 1, you can see another example of the use of lists. The response to the question *What language do you want to know more about?* can include more than one item from the list of menu items *[Pascal,C,Lisp,Prolog]*. The answers selected are stored as a list that is assigned to the topic *want*. Let's say that the answers selected were *C* and *Prolog*. When the command *do (?want).* is executed, each topic in the list assigned to *want* will be executed, so the first topic, *C*, will be performed and then the topic *Prolog* will be executed.

Lists enable you to manipulate groups of related items. In one knowledge base we examined recently, the designer had a long set of rules with each rule having only one conditional clause, as in this example:

```
if ?lot is K10
   then item is 3A62.

if ?lot is A22
   then item is 6H77.
...
topic lot.
ask ('What is the lot number on the
remaining section ?',lot,
   [K10,A22,...]).
end.
```

This entire set of rules, no matter what its length, could have been expressed in four lines using lists:

```
lot_list is [K10,A22,...].
item_list is [3A62,6H77,...].
ask ('What is the lot number ?',
                        lot,?lot_list).
item is element (?item_list,
   where (?lot_list, ?lot) ).
```

When the question is displayed, the contents of the topic *lot_list* are displayed on the menu. The item selected—for example, *6H77*—is assigned to the topic *lot*. The evaluation of the last line begins by finding where the value selected, *6H77*, is located in the list *lot_list*. The value *2* is returned as the result of the *where* command. Next, the *element* command returns the second item of the list contained in *item_list* and this value is assigned to the topic *item*.

Many other list-handling commands are necessary for a high-level symbolic language. These include commands such as *first*, *last*, and *rest* (similar to the LISP commands *car*, *last*, and *cdr*) that let you tear apart lists. KnowledgePro also includes commands for combining, comparing, and constructing lists in a variety of ways.

Summary

The flexibility of the hypertext and the directness that comes with procedural control make it possible for people with little or no programming experience to sit down and easily express their expertise. At the same time, techniques such as backward chaining, inheritance, and list handling enable it to be used in complex symbolic computational problems. With languages built around these types of structures, it becomes possible to design applications that allow program control to be shared by both the designer and the user of the system, surpassing the rigidity of the current expert system technology and the free-form environment of hypertext media.

Readers interested in seeing a sample of the KnowledgePro environment can obtain a run-time version from CompuServe: go PCVEN (data library 8). Several free demo programs, including TextPro (a program for writing and reading short hypertext documents), are available. Source code is included with the demos so you can get an idea of how topics work.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Theorem Proving Using Semantic Resolution

Resolving those nasty semantic clashes with C

by Anthony J. Dos Reis

One expects the *resolution principle*—a simple rule of inference useful in theorem proving—to date back to antiquity, along with *modus ponens* and Euclid's *elements*. In fact, though, it was discovered as recently as 1965 by J.A. Robinson and constituted a major breakthrough in mechanical theorem proving. It is surprising that such a powerful and (it seems now) obvious result did not surface much earlier.

Since 1965, the resolution principle has undergone many refinements. One such refinement—semantic resolution—is discussed in this article. To avoid becoming mired in the complexities of formal logic, I will restrict my attention to propositional logic. Although the power of propositional logic is limited, it is nevertheless an ideal vehicle for presenting the essential aspects of semantic resolution. (See Listing One page 64.)

Basic Definitions

I will use the symbols &, v, –>, and ¬ to represent the logical operations conjunction, disjunction, implication, and negation, respectively.

An expression (called the *conclusion*) is said to *logically follow* from a set of expressions (called the *premises*) if the conclusion is true whenever the premises are all true. A conclusion with its premises is called an *argument*. An argument is *valid* if its conclusion logically follows from its premises. The conclusion in a valid argument is called a *theorem*.

A *literal* is a variable or the negation of a variable. A *clause* is either a single literal or a disjunction of literals or is *empty* (denoted by the symbol [ ]). Examples of clauses are [ ], p, $\neg q$, $p \vee \neg q$, and $\neg p \vee q \vee r$. A variable and its negation (for example, p, $\neg p$) is called a *complementary pair*. The assignment of a truth value to every variable under consideration is called an *interpretation*. A set of clauses is *unsatisfiable* if for every interpretation at least one clause in the set is false.

Expressions with identical truth table values are said to be *equivalent*. It can be shown that any logical expression can be transformed to an equivalent expression consisting of either a single clause or a conjunction of clauses. For example, the expression $(p \vee q) \rightarrow r$ can be converted to the equivalent expression $(\neg p \vee r) \wedge (\neg q \vee r)$. Thus, without loss of generality, we can restrict our attention to logical expressions of this standard form. In fact, we can restrict our attention to individual clauses because a conjunction can be viewed as several individual premises or conclusions. For example, the premise $(\neg p \vee r) \wedge (\neg q \vee r)$ can be viewed as two premises—$(\neg p \vee r)$ and $(\neg q \vee r)$—each a single clause.

To prove that a conclusion logically follows from its premises in propositional logic, you simply need to construct a truth table and determine if the conclusion meets the definition of logically follows. Although this approach is eminently satisfactory for propositional logic, it unfortunately falls apart in more powerful logic systems (such as first-order logic) in which truth tables may not be of finite size. For such systems rules of inference are used to establish whether a conclusion logically follows from its premises. A *rule of inference* is a rule that describes how to generate a new expression that logically follows from existing expressions. The step-by-step derivation of the conclusion from the premises using rules of inference is called a *proof*.

Typically, you need many rules of inference to generate the expressions needed in a proof. For example, the rule of inference *modus ponens* given by:

```
E1 –> E2
E1
∴ E2
```

Anthony J. Dos Reis is an assistant professor at the College at New Paltz, Dept. of Mathematics and Computer Science, New Paltz, NY 12561.

cannot be used to show that $\neg p$ logically follows from $p \rightarrow q$ and $\neg q$ because the premises are not of the appropriate form. *Modus ponens* by itself is not *complete*—that is, there are valid arguments whose conclusions cannot be generated using *modus ponens* alone.

A serious problem with having many rules of inference is that it is necessary to select the appropriate rule at each step in a proof. The appropriate rule is, in general, not obvious. You could, of course, try every rule at every step, but such a brute-force approach is very inefficient in time and space.

The Resolution Principle

The resolution principle is a rule of inference given by the four forms shown in Table 1, below.

Resolution requires two clauses (the clauses to be resolved) with a complementary pair—that is, one clause should contain some expression *E1* and the other $\neg E1$. The *resolvent*—that is, the clause generated—is formed from the expressions that remain after *E1* and $\neg E1$ are thrown away. If more than one expression remains (as in the first form in Table 1), the resolvent is the disjunction of these expressions. If no expressions remain (as in the fourth form in Table 1), the resolvent is the empty clause, denoted by *[ ]*. Because disjunction is commutative, the order of the expressions is not important.

If two clauses that both contain some expression E are resolved, the resolvent need not have two occurrences of E because E v E always equals E. For example, you can take the resolvent of p v q v r and ¬p v q to be q v r rather than q v q v r.

If clauses that have more than one complementary pair are resolved, only one complementary pair is eliminated. For example, if you resolve $p \vee q$ and $\neg p \vee \neg q$, you get either $q \vee \neg q$ or $p \vee \neg p$, depending on which complementary pair you eliminate. Here the resolvent is always a tautology (that is, an expression that is always true) and therefore logically follows from any set of premises. In refutation proofs (see later), such resolutions are useless and should never be performed.

In the following example, 6 through 9 are obtained by resolution. Each is labeled with the line numbers of the expressions from which it is derived.

```
1 p v ¬q v r
2 p v q v s
3 ¬p
4 ¬r
5 ¬s
6 p v r v s      (1,2)
7 p v r          (5,6)
8 r              (3,7)
9 [ ]            (4,8)
```

The rule of inference *modus ponens* given in clause form:

```
¬E1 v E2
  E1
∴ E2
```

is clearly a special case of the resolution principle. Other rules of inference—*modus tollens*, *hypothetical syllogism*, *disjunctive syllogism*, *constructive dilemma*, and *destructive dilemma*—are also special cases of resolution. That resolution encompasses all these rules is an indication of its power and generality.

Proof by Refutation

Suppose you wish to show that the argument:

```
  P1
  P2
   .
   .      premises
   .
  Pn
∴ C      conclusion
```

is valid—that is, *C* logically follows from *P1*, *P2*, . . ., *Pn*. The straightforward approach is to derive C from the premises using rules of inference. Another approach—called proof by refutation—is to negate the conclusion and then derive the empty clause from the premises and the negated conclusion. I'll first look at an example and then discuss why this approach works.

To show that the following argument is valid:

```
  ¬p v r
  ¬r v t      premises
    p
∴ T           conclusion
```

first negate the conclusion and add it to the premises. Then using rules of inference, derive the empty clause:

```
1 ¬p v r
2 ¬r v t
3 p
4 ¬t      negation of the conclusion
5 r       (1,3)
6 ¬r      (2,4)
```

(i)	(ii)	(iii)	(iv)
E1 v E2	E1 v E2	E1	E1
¬E1 v E3	¬E1	¬E1 v E3	¬E1
∴E2 v E3	∴E2	∴E3	∴[]

Table 1: *The four forms of resolution*

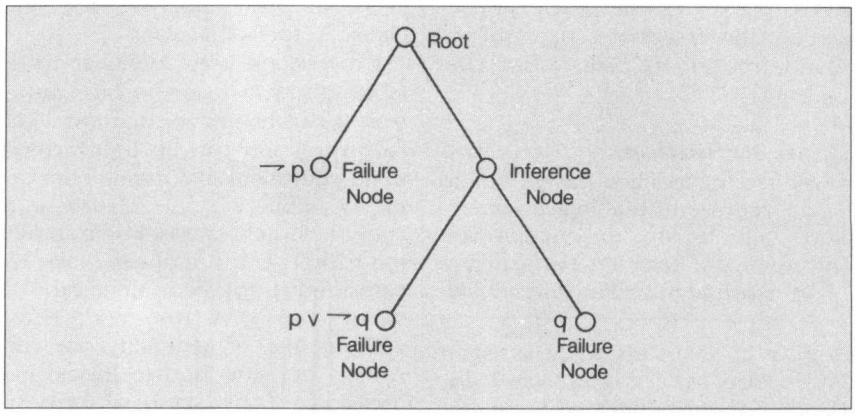


Figure 1: *Binary tree*

SEMANTIC RESOLUTIONS
(continued from page 51)

7 [ ] (5,6)

To understand why this approach works, first note that the generation of the empty clause (line 7) requires the resolution of an expression and its negation, r and $\neg r$. Now assume that the initial clauses (1, 2, 3 and 4) are all true. Then both r and $\neg r$ must also be true because they are derived from the initial clauses using a rule of inference. But because r and $\neg r$ always have opposite truth values, they cannot both be true. The assumption—that the initial clauses are all true—must be false. Thus, whenever clauses 1, 2, and 3 are all true, clause 4 (the negation of the conclusion) is false and the conclusion is true.

Although proof by refutation seems like a roundabout way of proving a theorem, it is actually more suitable for machine implementation than the straightforward approach for two reasons. First, the goal of the refutation approach is an empty clause, which is generally easier to derive because it may be obtained from any one of perhaps several complementary pairs. For example, another possibility for the previous proof is to derive $\neg t$ and t, as follows:

1 ¬p v r
2 ¬r v t
3 p
4 ¬t negation of the conclusion
5 r (1,3)
6 t (2,5)
7 [ ] (4,6)

Second and even more important, resolution is complete when used in a refutation proof—that is, if the original argument is valid, then the empty clause can always be derived using only the resolution principle. Our bagful of inference rules is not needed.

Completeness of Resolution

The basic idea in the proof of the completeness of the resolution principle can be illustrated by considering a simple example. Suppose the set of clauses from which the empty clause is to be derived is $\{\neg p, p \vee q, \neg q, q\}$. Construct a binary tree as follows:

1. Start with an unlabeled root node.

Now repeat steps 2 and 3 for each variable V that appears among the set of clauses (in this example, steps 2 and 3 will be done twice—once for $V=p$ and once for $V=q$):

2. Extend the tree from any unlabeled leaf node by adding two branches that terminate on two new nodes, the left and right of which represent the assignment of true and false, respectively, to V.
3. If the truth assignment associated with all the nodes in the path from the root to a node added in step 2 makes a clause false, then label that node with that clause. (Labeled nodes are called failure nodes.)

The tree constructed corresponding to this example is shown in Figure 1, page 51. Observe that every leaf node must be a failure node because otherwise an interpretation would exist that satisfies the set of clauses. Moreover, at least one node (called an inference node) must have two failure nodes immediately below it. Because the truth assignments associated with these failure nodes differ for exactly one variable, their corresponding clauses must contain exactly one complementary pair and are therefore resolvable. Furthermore, the resolvent is false (in the sense of step 3 above) for the inference node. Thus, the inference node can be labeled with the resolvent (making it a failure node) and the two nodes below it pruned. Thus, you get the tree shown in Figure 2, page 55.

For resolution to be practical, it must be restricted from generating too many unnecessary resolvents

The same argument can be applied repeatedly, resulting in further pruning (and resolution) until only the root node remains, corresponding to the derivation of the empty clause.

Semantic Resolution

In a resolution proof, the clauses to be resolved must be selected at each step. In general, there may be many resolvable clauses that produce resolvents that are not required for the proof. For example, consider:

1 p v q
2 p v ¬q
3 ¬p v q
4 ¬p v ¬q
5 p (1,2)

6 →p (3,4)
7 [ ] (5,6)

The resolvents *(1,3)*, *(2,4)*, *(1,6)*, *(2,6)*, *(3,5)*, and *(4,5)* are not needed for the proof and are, therefore, wasteful to generate. A mechanical theorem prover, however, unless otherwise constrained, would typically generate such resolvents. For resolution to be practical, it must be restricted from generating too many unnecessary resolvents. The restrictions used, however, must still allow the necessary resolvents to be generated, or else the technique would not be complete.

One approach to restricting resolution that is complete is called *semantic resolution*. Semantic resolution applies two restrictions to resolution: one using an interpretation; the other, an ordering. Recall that an *interpretation* is the assignment of a truth value to each variable. In the following set of clauses, for example:

1 p v q
2 p v →q
3 →p v q
4 →p v →q

p can be assigned true and *q* false. With this particular interpretation, clauses 1, 2, and 4 all evaluate to true, and clause 3 evaluates to false. For a given interpretation, a clause that evaluates to true is called a *nucleus* (denoted by a +), and a clause that evaluates to false is called an *electron* (denoted by a –). An *ordering* is simply a listing, in descending priority, of the variables in a set of clauses. A single interpretation and a single ordering should be used for the entire proof. The particular interpretation and ordering used is not critical, however—any will work.

The two restrictions of semantic resolution are:

1. Never resolve a nucleus with a nucleus.
2. Resolve an electron with a nucleus only if the variable to be eliminated has the highest priority among the variables that appear in the electron.

An example should make these two rules clear. For an interpretation, assign true to *p* and *q* and take <*p*, *q*> as the ordering. Now consider the following clauses:

1 p v q (+)
2 p v →q (+)
3 →p v q (+)
4 →p v →q (–)

The possible resolvents are *(1,2)*, *(1,3)*, *(2,4)*, and *(3,4)*. By restriction 1, *(1,2)* and *(1,3)* should not be generated because both are nuclei. By restriction 2, *(3,4)* should not be generated because the variable *p* appears in the electron and has higher priority than *q*, the variable eliminated. Thus, the only allowable resolvent is *(2,4)*:

5 →q (2,4), (–)

With clause 5 added to the list, the only possibilities now are *(1,5)* and *(3,5)*:

6 p (1,5), (+)
7 →p (3,5), (–)

Now the possibilities are *(4,6)*, *(1,7)*, *(2,7)*, and *(6,7)*:

8 →q (4,6), (–)
9 q (1,7), (+)
10 →q (2,7), (–)
11 [ ] (6,7)

Two additional points should be

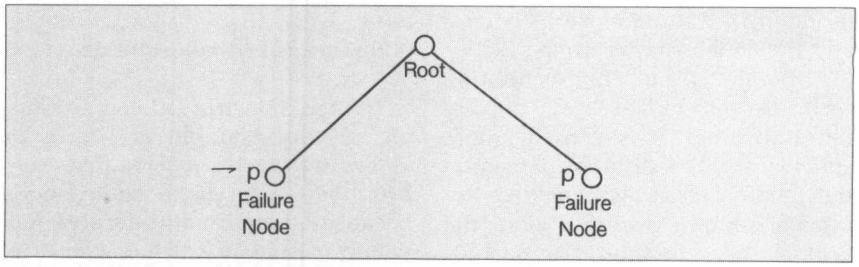


Figure 2: *Binary tree after resolution*

SEMANTIC RESOLUTIONS
(continued from page 55)

made about semantic resolution. First, if an interpretation makes the clauses all true or all false, then the original argument is not valid and, therefore, cannot be proved. Second, an electron can never be resolved with an electron because resolution requires a pair of clauses with complementary expressions, one of which must be true regardless of the interpretation used. Thus, one of the clauses is always a nucleus.

Semantic Clashes

Consider the following proof using semantic resolution with true assigned to *p*, *q*, and *r* as the interpretation and with <*p*, *q*, *r*> as the ordering. All the resolvents that can be generated under semantic resolution are listed.

```
1 p v ¬q v r          (+)
2 q                   (+)
3 ¬p v ¬q             (–)
4 ¬r                  (–)
5 ¬q v r        (1,3) (+)
6 p v ¬q        (1,4) (+)
7 ¬q            (4,5) (–)
8 ¬q            (3,6) (–)
9 [ ]           (2,7)
```

The true literals, *p* and *r*, in the nucleus on line 1 can be eliminated, respectively, by the electron on line 3 and the electron on line 4. Clauses 1, 3, and 4 constitute a *semantic clash*—that is, a single nucleus together with a set of electrons that eliminate all its true literals under semantic resolution. A semantic clash contains exactly one electron for each true literal in its nucleus.

The nucleus and electrons in a semantic clash can be resolved in the following manner. First, resolve the nucleus with any electron in the semantic clash. Then resolve the resulting resolvent with another electron. Continue resolving electrons with successive resolvents until all the electrons are used. For example, for the semantic clash *(1,3,4)* above, you get:

```
1 p v ¬q v r          (+)
2 q                   (+)
3 ¬p v ¬q             (–)
4 ¬r                  (–)
5 ¬q v r        (1,3) (+)
intermediate resolvent
6 ¬q            (4,5) (–)
final resolvent
```

The final resolvent of a semantic clash is always an electron or the empty clause—never a nucleus.

The importance of resolving semantic clashes is that the intermediate resolvents are never needed to produce the empty clause and therefore can be omitted. Fewer unnecessary resolvents are generated, resulting in a more efficient proof. The previous proof with nine lines, redone with semantic clashes, becomes a proof with only six lines:

```
1 p v ¬q v r (+)
2 q (+)
3 ¬p v ¬q (–)
4 ¬r (-)
5 ¬q (1,3,4) (–) semantic clash
6 [ ] (2,5) semantic clash
```

A semantic clash does not necessarily exist for every nucleus. For example, a semantic clash does not exist for clause 2 above until clause 5 is generated. However, because resolution restricted to semantic clashes is complete (for a proof, see Chang and Lee), there should always be at least one semantic clash until the empty clause is generated.

Bibliography

Chang, C.; and Lee, R.C. *Symbolic Logic and Mechanical Theorem Proving*. New York, N.Y.: Academic Press, 1973.

Winston, P.H. *Artificial Intelligence*. 2d ed. Englewood Cliffs, N.J.: Addison-Wesley, 1984.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 64.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Listing One *(Text begins on page 18.)*

```
Listing One.  A simple adventure game written in AAL

(loc the-first-room
"You are in a small, gloomy room lit by an unseen source above you.
The walls and floor are smooth, hard and dark, like obsidian.  Exits
lead west and south."
  (contains whistle)
  (exits
   (w the-second-room)
   (s "You have wandered around and wound up back where you started")))


(loc the-second-room
"You are in a vast chamber of ice and rock.  Fiery torches in the walls
provide an eerie light.  There is a passageway south and another exit to
the north."
  (contains monster)
  (exits
   (s "The passageway is blocked by rubble.")
   (n (((alive monster) -> "The monster won't let you pass.")
       the-first-room))))

(command blow
        (blow *obj)
        (requires ((carrying player *obj) "You don't have a" *obj))
        "You can't blow that!")

(command (throw hurl chuck)
        (throw *instr at *obj)
        (requires (carrying player *instr)
                  (here *obj))
        "Nothing happens."))

(obj monster fixed
     (action throw *obj
             ("The monster destroys the a" *instr)
             (destroy *instr)))

(obj whistle
     (action blow *obj
             "The whistle emits a piercing screech."
             ((here monster) ->
              "The monster's eyes bug out--wider--wider--and then,
                finally, close forever."
              (dead monster))))
```

End Listing One

Listing Two

```
Listing 2 for Amsterdam's DDJ article: The deducer

(defun assert (list)
  (let ((new-stmt (list->rule list)))
    (if (and (add-to-database new-stmt) (eq (rule-type new-stmt) :fact))
        (run-forward-rules *assertion-rules* new-stmt)))
    new-stmt)

(defun retract (list)
  (let ((new-stmt (list->rule list)))
    (when (remove-from-database new-stmt)
      (run-forward-rules *retraction-rules* new-stmt))
    new-stmt))

(defun run-forward-rules (rules fact)
  ;; Run a rule if its antecedent matches the fact.  (Forward rules can
  ;; only have one antecedent.)
  (dolist (frule rules)
    (let ((bindings (unify fact (car (rule-antecedents frule)) *globals*)))
      (if (not (eq bindings :fail))
          (execute-action (rule-consequents frule) bindings)))))


(defun deduce (pattern-list bindings)
  ;; Returns a stream of bindings (variable lists) for things that match all
  ;; the patterns, or the empty stream if there are none.
  (if (null pattern-list)
      (stream-cons bindings *empty-stream*)
      (let ((bindings-stream (deduce-pattern (car pattern-list) bindings)))
        (stream-mapcan #'(lambda (b) (deduce (cdr pattern-list) b))
                       bindings-stream))))

(defun deduce-pattern (pattern bindings)
  (deduce-pat pattern bindings (find-possible-unifiers pattern)))

(defun deduce-pat (pattern bindings possibilities)
  (if (null possibilities)
      *empty-stream*
      (let* ((rule-pats (rename-rule (car possibilities)))
             (bindings1 (unify pattern (car rule-pats) bindings)))
        (if (eq bindings1 :fail)
            (deduce-pat pattern bindings (cdr possibilities))

            (stream-append
              (deduce (cdr rule-pats) bindings1)
              (deduce-pat pattern bindings (cdr possibilities)))))))


(defun unify (pat1 pat2 bindings)
  ;;Returns :FAIL if it can't unify, a list of bindings if it can.
  (cond
    ((and (null pat1) (null pat2))
     bindings)
    ((or (null pat1) (null pat2))
     :fail)
    ((let* ((el1 (car pat1))
            (el2 (car pat2))
            (new-bindings (if (var? el1)
                              (unify-var el1 el2 bindings)
                              (unify-const el1 el2 bindings))))
       (if (eq new-bindings :fail)
           :fail
           (unify (cdr pat1) (cdr pat2) new-bindings))))))

(defun unify-var (v el bindings)
  (let ((val (var-value v bindings)))
    (if (eq val :unbound)
        (cons (cons v el) bindings)
        (unify-const val el bindings))))

(defun unify-const (const el bindings)
  (if (var? el)
      (unify-var el const bindings)
      (if (eql const el) bindings :fail)))

(defun find-possible-unifiers (pattern)
  ;; Returns a list of rules and facts that might unify with pattern
  (if (var? (car pattern))
      *db*
      (append  (get '* 'database)
               (get (car pattern) 'database))))

(defun add-to-database (rule)
  ;; Returns NIL iff not added (because already present)
  (let ((index (index-of rule)))
    (cond
      ((member rule (get index 'database) :test #'equalp)
       nil)
      (t
       (push rule (get index 'database))
       (push rule *db*)
       t))))
(defun remove-from-database (rule)
  ;; Returns NIL iff not removed (because not present)
  (let* ((index (index-of rule))
         (the-rule (car (member rule (get index 'database) :test #'equalp))))
    (cond
      (the-rule
       (setf (get index 'database) (delete the-rule (get index 'database)
                                           :test #'eq))
       (setq *db* (delete the-rule *db* :test #'eq))
       t)
      (t nil))))

(defun index-of (rule)
  (caar (rule-consequents rule)))

(defun var? (thing)
  (and (symbolp thing) (char= (char (symbol-name thing) 0) #\*)))

(defun var-value (var bindings)
  ;; Follows the chain of variables bindings to find the ultimate value of var.
  (let ((val-pair (assoc var bindings)))
    (if (not val-pair)
        :unbound
        (let ((val (cdr val-pair)))
          (if (var? val)
              (var-value val bindings)
              val)))))

(defun rename-rule (rule)
  ;; Returns a list (c . a) of consequent and antecedents, with all
  ;; variables renamed.
  (multiple-value-bind (ants bindings)
      (copy-pattern-list (rule-antecedents rule) nil)
    (multiple-value-bind (conseqs ignorable-bindings)
        (copy-pattern-list (rule-consequents rule) bindings)
      (cons (car conseqs) ants))))

(defun copy-pattern-list (pat-list bindings)
  (if (null pat-list)
      (values nil bindings)
      (multiple-value-bind (new-pat binds)
          (copy-pattern (car pat-list) bindings)
        (multiple-value-bind (rest-pat final-binds)
            (copy-pattern-list (cdr pat-list) binds)
          (values (cons new-pat rest-pat) final-binds)))))

(defun copy-pattern (pat bindings)
  (do* ((els pat (cdr els))

        (el (car els) (car els))
        (new-pat nil))
       ((null els) (values (nreverse new-pat) bindings))
    (if (not (var? el))
        (push el new-pat)
        (let ((existing-var-pair (assoc el bindings)))
          (if existing-var-pair
              (push (cdr existing-var-pair) new-pat)
              (let ((new-var (rename-var el)))
                (push new-var new-pat)
                (push (cons el new-var) bindings)))))))

(defun rename-var (var)
  ;; Generate a new symbol.
  (gentemp (symbol-name var)))
```

End Listing Two

Listing Three

```
Listing Three. Code for streams

(defvar *empty-stream* nil)

(defmacro delay (thing)
  `#'(lambda () ,thing))

(defun force (thing)
  (funcall thing))

(defmacro stream-cons (thing stream)
  `(cons ,thing (delay ,stream)))

(defun stream-empty? (stream)
  (eq stream *empty-stream*))

(defun stream-car (stream)
  (car stream))

(defun stream-cdr (stream)
  (force (cdr stream)))

(defmacro dostream ((var stream) &body body)
  (let ((tempvar (gensym)))
    `(do* ((,tempvar ,stream (stream-cdr ,tempvar))
           (,var (stream-car ,tempvar) (stream-car ,tempvar)))
          ((stream-empty? ,tempvar) *empty-stream*)
       ,@body)))

(defmacro stream-append (stream1 stream2)
  `(stream-append-func ,stream1 (delay ,stream2)))


(defun stream-append-func (stream delayed-stream)
  (if (stream-empty? stream)
```

```
      (force delayed-stream)
      (stream-cons (stream-car stream)
                   (stream-append-func (stream-cdr stream) delayed-stream))))

(defun stream-mapcar (function stream)
  (if (stream-empty? stream)
      *empty-stream*
      (stream-cons (funcall function (stream-car stream))
                   (stream-mapcar function (stream-cdr stream)))))

(defun stream-mapcan (function stream)
  (if (stream-empty? stream)
      *empty-stream*
      (stream-append (funcall function (stream-car stream))
                     (stream-mapcan function (stream-cdr stream)))))

(defun stream->list (stream)
  (if (stream-empty? stream)
      nil
      (cons (stream-car stream)
            (stream->list (stream-cdr stream)))))

(defun list->stream (list)
  (if (null list)
      *empty-stream*
      (stream-cons (car list)
                   (list->stream (cdr list)))))
```

End Listing Three

Listing Four

```
Listing Four.  Code for the every action

(defun do-every-action (rule bindings)
  ;; Get a list of bindings for the single quantified variable, using the
  ;; antecedents; then execute the consequents for each binding.
  (let* ((quant-vars (rule-quant-vars rule)))
    (if (not (= (length quant-vars) 1))
        (error "Only one quantified variable allowed in rule a" rule)

      (let* ((bindings-stream (deduce (rule-antecedents rule) bindings))
             (bindings-list (stream->list bindings-stream))
             (filtered-list (mapcar #'(lambda (b) (extract-bindings b
                                                    quant-vars))
                                    bindings-list))
             (undup-list (delete-duplicate-bindings filtered-list))
             (new-bindings-list (mapcar #'(lambda (b) (append b bindings))
                                        undup-list)))
        (dolist (new-bindings new-bindings-list)
          (do-rule-actions (rule-consequents rule) new-bindings))))))
```

End Listing Four

Listing Five

```
Listing Five.  The check-reqs function

(defun check-reqs (reqs bindings)
  (if (null reqs)
      t
      (let* ((req (car reqs))
             (binding-stream (deduce-pattern (requirement-pattern req)
                                             bindings))
             (fstring nil))
        (cond
          ((stream-empty? binding-stream)
           (return-from check-reqs (if (requirement-succeeded? req)
                                       nil
                                       (requirement-failure-string req))))
          (t
           (setf (requirement-succeeded? req) t)
           (dostream (binds binding-stream)
             (let ((result (check-reqs (cdr reqs) binds)))
               (if (eq result t)
                   (return-from check-reqs t)
                   (if result
                       (setq fstring result)))))
           fstring)))))
```

End Listings

SEMANTIC RESOLUTIONS

Listing One *(Text begins on page 50.)*

```
/*      A Theorem Prover for Propositional Logic
 *
 *      This program uses the resolution principle restricted to
 *  semantic clashes to prove theorems in propositional logic.
 *  The premises and the negation of the conclusion must be entered
 *  using 1, 0, and -1 to represent, respectively, the presence of a
 *  variable in true form, the absence of a variable, and the
 *  presence of a variable in negated form.  For example, a run
 *  in which the following four clauses are used
 *
 *       p v *q v  r
 *            q
 *      *p v *q
 *                *r
 *
 *  would look as follows:
 *
 *  enter number of clauses and variables
 *  4 3
 *  enter clauses
 *  1  -1  1
 *  0   1  0
 * -1  -1  0
 *  0   0 -1
 *
 *  The program would then respond with
 *
 *  initial clauses
 *    1 -1  1       (1)
 *    0  1  0       (2)
 *   -1 -1  0       (3)
 *    0  0 -1       (4)
 *  resolvents
 *    0 -1  0       (5 from 1, 3, 4)
 *    0  0  0       (6 from 2, 5)
 *  proof complete--empty clause generated
 *
 *      For each nucleus, the program searches for a set of electrons
 *  that make up a semantic clash.  When it finds a semantic clash, it
 *  generates the resolvent and then uses a recursive technique called
 *  backtracking to find other sets of electrons that also make up
 *  a semantic clash.  This implementation is compact but quite
 *  slow.
 *
 *  Author:  A. J. Dos Reis
 *           Dept. of Mathematics and Computer Science
 *           College at New Paltz
 *           New Paltz, N.Y. 12561
 *-------------------------------------------------------------------
 *     CONSTANTS
 */
#define TRUE 1
#define FALSE 0
#define MAXCLAUSES 100
#define MAXVARS 10
#define ELECTRON -1
#define NUCLEUS 1
/*-------------------------------------------------------------------
 *     GLOBAL VARIABLES
 */
int clause[MAXCLAUSES][MAXVARS];     /* Holds clauses              */
int clausetype[MAXCLAUSES];          /* Holds type:  nuc or elec   */
int clashelec[MAXVARS];              /* Holds row numbers of electrons
                                        that form a semantic clash  */
int nvars;                           /* Number of variables        */
int avail;                           /* Index of next avail row    */
int newclauses;                      /* TRUE if new resolvents
                                        generated on last pass     */
int noemptyclause;                   /* TRUE if empty clause not
                                        generated                  */
/*-------------------------------------------------------------------
 *     main prompts for and inputs the clauses.  It then calls
 *     findsemclash for each nucleus.  It continues until the
 *     empty clause is generated or until no new resolvents are
 *     generated.
 */
main()
{
   int nclauses;                     /* Number of clauses          */
   int nuc;                          /* Index of nucleus           */
   int i,j;                          /* Utility indices            */
   printf("enter number of clauses and variables\n");
   scanf("%d%d",&nclauses,&nvars);
   printf("enter clauses\n");
   for (i=0; i<nclauses; i++)        /* Read in clauses            */
      {clausetype[i]=ELECTRON;
       for (j=0; j<nvars; j++)
          {scanf("%d",&clause[i][j]);
           if (clause[i][j]==1)
              clausetype[i]=NUCLEUS; /* Set clausetype accordingly */
          }
      }
   avail=nclauses;                   /* Keep track of next avail row*/
   printf("initial clauses\n");      /* Print out initial clauses  */
   for (i=0; i<nclauses; i++)
      {for (j=0; j<nvars; j++)
         printf("%3d",clause[i][j]);
       printf("     (%ld)\n",i+1);
      }
   printf("resolvents\n");
   noemptyclause=TRUE;               /* Initialize flag            */
   do
      {newclauses=FALSE;             /* Initialize flag            */
       nuc=0;                        /* Start search from row 0    */
       do
          {if (clausetype[nuc]==NUCLEUS) /* Search for nucleus      */
             {for (i=0; i<nvars; i++) /* Initialize clashelec       */
                clashelec[i]=-99;
              findsemclash(0,nuc);    /* Look for all sem clashes   */
             }
           nuc++;                     /* Repeat for next clause     */
          } while ((nuc<nclauses) && noemptyclause);
      } while (newclauses && noemptyclause);
   if (noemptyclause)                /* Failed to gen empty clause? */
      printf("argument not valid\n");
   else
      printf("proof complete--empty clause generated\n");
}
/*-------------------------------------------------------------------
 *     findsemclash searches for an electron which under semantic
 *     resolution will eliminate the true literal in column truecol
 *     of the nucleus in row nuc.  When it finds a satisfactory
 *     electron, it recursively calls itself to find a electron that
 *     eliminates the next true literal.  When a set of electrons
 *     that forms a semantic clash is found, findsemclash generates
 *     the resolvent instead of making a recursive call.  Then
 *     it backtracks to find other sets that also form semantic clashes.
 */
findsemclash(truecol,nuc)
{int col;                            /* Column index               */
 int elec;                           /* Row number of electron     */
 int goodelec;                       /* True if elec ok under
                                        semantic resolution        */
 int i,j;                            /* Utility indices            */
 while (truecol<nvars)               /* Checked all columns of nuc? */
   if (clause[nuc][truecol]==1)      /* Is this a true literal     */
      {elec=0;                       /* Search for appropriate elec */
       do
          {if (clausetype[elec]==ELECTRON)
              {col=0;                /* check col-by-col if elec ok */
               goodelec=TRUE;
               while ((col<nvars) && goodelec)
                  {if (               /* Satisfies sem resolution?  */
                        ( (col<truecol) && (clause[elec][col]!=0))
                                      ||
                        ( (clause[nuc][col] * clause[elec][col]==-1)
                                      !=
                          (truecol==col))
                      )
                     goodelec=FALSE; /* Electron fails test        */
                   col++;             /* Test next column           */
```

```
                }
             if (goodelec)          /* Electron passed test?       */
                {clashelec[truecol]=elec; /* Remember row number  */
                 findsemclash(truecol+1,nuc); /* Recursive call   */
                }
           }
         elec++;                  /* Try next electron           */
       } while ((elec<avail) && noemptyclause);
     return;                      /* Backtrack                   */
    }
  else
    truecol++;                    /* Continue search for true lit
                                     in the nucleus              */
 newclauses=TRUE;                 /* Get here only when a sem
                                     clash is found              */
 for (i=0; i<nvars; i++)          /* Copy nuc to new avail row   */
   clause[avail][i]=clause[nuc][i];
 for (i=0; i<nvars; i++)
   {elec=clashelec[i];            /* Get row number of electron  */
    if (elec!=-99)                /* -99 means electron not needed
                                     for the ith col             */
      for (j=0; j<nvars; j++)     /* Generate the resolvent      */
        {clause[avail][j]+=clause[elec][j];
         if (clause[avail][j]!=0)
           noemptyclause=TRUE;
         if (clause[avail][j]==-2)
           clause[avail][j]=-1;
        }
   }
 noemptyclause=FALSE;             /* Initialize flag             */
 for (j=0; j<nvars; j++)          /* Check for empty clause      */

   {printf("%3d",clause[avail][j]);
    if (clause[avail][j])
      noemptyclause=TRUE;         /* Reset if no empty clause    */
   }
 printf("     (%ld from %ld",avail+1,nuc+1); /* Print resolvent  */
 for (i=0;i<nvars; i++)
   if (clashelec[i]!=-99)         /* Print electron row numbers */
     printf(", %ld",clashelec[i]+1);
 printf(")\n");
 clausetype[avail]=ELECTRON;      /* Set type of resolvent       */
 avail++;                         /* Increment avail row index  */
 }
```

End Listing

C CHEST

Listing One *(Text begins on page 98.)*

```
 1| #include <stdarg.h> /* ANSI variable-argument defines          */
 2| #include <dos.h>    /* uSoft: defines for FP_OFF and FP_SEG    */
 3|
 4| /* IDOPRNT - integer-only printf/fprintf/sprintf workhorse
 5| *            function.
 6| *
 7| *            (C) 1988 Allen I. Holub. All rights reserved.
 8| *
 9| * The following conversions are supported:
10| *
11| *      %d %ld  decimal, long decimal
12| *      %u      unsigned (int only, no longs)
13| *      %s      string
14| *      %x %lx  hex, long hex
15| *      %o %lo  octal, long octal
16| *      %b %lb  binary, long binary                    (nonstandard)
17| *      %p      far pointer (in hex XXXX:XXXX)
18| *
19| *      Note that it's impossible to get zero fill in the offset
20| *      part of a %p conversion. That is 1234:0001 is always
21| *      printed as 1234:1. Sorry. The following modifiers are
22| *      supported for all conversions:
23| *
24| *      %0x     zero left fill
25| *      %-x     left justification in field
26| *      %|x     centered in field               (nonstandard)
27| *      %*x     field width from first argument
28| *
29| *      Precision is supported in strings:
30| *
31| *      %X.Ys   X-character wide field, print at most Y characters
32| *              (even if the string is longer). If X or Y is a *,
33| *              the width is taken from the next argument.
34| *
35| *      Potential portability problems:
36| *
37| *      %p is a FAR pointer. I've used the "far" keyword to
38| *      declare it as such and I've used the FP_SEG and FP_OFF
39| *      to extract the segment and offset parts of the pointer.
40| *      The offset part of a near pointes can be printed by
41| *      casting it to an int and using %x.
42| */
43|
44| extern char     *ltos(long, char*, int);
45|
46| /*------------------------------------------------------------
47|  * Macros to save some code space below. If your compiler
48|  * doesn't accept multi-line macros, remove the backslashes
49|  * and merge the entire definition onto one line.
50|  * These both have horrible side effects. Be careful.
51|  *
52|  * PAD(fw,filchar,out,op)   outputs filchar, fw times. fw = 0.
53|  * TOINT(p,x)               works like "x = atoi(p);" except p
54|  *                              is advanced past the number.
55|  */
56|
57| #define PAD(fw,fc,out,op)     while( --(fw) >= 0 )        \
58|                                   (*out)( fc, op )
59|
60| #define TOINT(p,x)  while( '0' <= *p  &&  *p <= '9' )    \
61|                         x = (x * 10) + (*p++ - '0')
62|
63| /*------------------------------------------------------------
64|  * INTMASK is a portable way to mask off the bottom N bits
65|  * of a long, were N is the width of an int.
66|  */
67|
68| #define INTMASK         (long)( (unsigned)(0) )
69|
70| /*------------------------------------------------------------*/
71|
72| void    idoprnt( out, o_param, format, args )
73|
74| int     (*out)();       /* output subroutine                */
75| void    *o_param;       /* 2nd argument to pass to out()    */
76| char    *format;        /* pointer to format string         */
77| va_list args;           /* pointer to arguments             */
78| {
79|     char filchar ;      /* Fill character used to pad fields */
80|     char nbuf[34];      /* Buffer used to hold converted #s  */
81|     char *bp     ;      /* Pointer to current output buffer  */
82|     int slen     ;      /* length of string pointe to by bp  */
83|     int base     ;      /* Current base (%x=16, %d=10, etc.) */
84|     int fldwth   ;      /* Field width as in %10x            */
85|     int precision;      /* Precision as in %10.10s or %10.3f */
86|     int lftjust  ;      /* 1 = left justifying (ie. %-10d)   */
87|     int center   ;      /* 1 = centered (ie. %|10d)          */
88|     int longf    ;      /* doing long int (ie. %lx or %X)    */
89|     long lnum    ;      /* used to hold numeric arguments    */
90|     void far *pnum;     /* Pointer-sized number              */
91|
92|     for(; *format ; format++ )
93|     {
94|         if( *format != '%')           /* No conversion, just  */
95|         {                             /* print the next char. */
96|             (*out)(*format, o_param);
97|         }
98|         else                          /* Process a % conversion */
99|         {
100|            bp          = nbuf  ;
101|            filchar     = ' '   ;
102|            fldwth      = 0     ;
103|            lftjust     = 0     ;
104|            center      = 0     ;
105|            longf       = 0     ;
106|            precision   = 0     ;
107|            slen        = 0     ;
108|
109|            /*  Interpret any modifiers that can precede a
110|             *  conversion character. (ie %04x , %-10.6s... etc).
111|             *  if a * is present instead of a field width then
112|             *  the width is taken from the argument list.
113|             */
114|
115|            if( *++format == '-') { ++format ;  ++lftjust;     }
116|            if( *format   == '|') { ++format ;  ++center;      }
117|            if( *format   == '0') { ++format ;  filchar = '0'; }
118|
119|            if( *format != '*' )
120|                TOINT( format, fldwth );
121|            else
122|            {
123|                ++format;
124|                fldwth = va_arg(args, int);
125|            }
126|
127|            if( *format == '.' )
128|            {
129|                if( *++format != '*' )
130|                    TOINT( format, precision );
131|                else
132|                {
133|                    ++format;
134|                    precision = va_arg(args, int);
135|                }
136|            }
137|
138|            if( *format == 'l' || *format == 'L' )
139|            {
140|                ++format;
141|                ++longf ;
142|            }
143|
144|
145|            /* By now we've picked off all the modifiers and
146|             * *format is looking at the actual conversion
147|             * character. Pick the appropriatly sized argument
148|             * off the stack and advanced the pointer (ap) to
149|             * point at the next argument.
150|             */
151|
152|            switch( *format )
153|            {
154|            default:    *bp++ = *format ;               break;
155|            case 'c':   *bp++ = va_arg(args, int   );   break;
156|            case 's':   bp    = va_arg(args, char *);   break;
157|
158|            case 'p':
159|                pnum  = va_arg(args, void far *);
160|                bp    = ltos( (unsigned long)FP_SEG(pnum), bp, 16);
161|                *bp++ = ':';
162|                bp    = ltos( (unsigned long)FP_OFF(pnum), bp, 16 );
163|                break;
164|
165|            case 'u':   base = -10 ; goto pnum ;
166|            case 'd':   base = 10  ; goto pnum ;
167|            case 'x':   base = 16  ; goto pnum ;
168|            case 'b':   base =  2  ; goto pnum ;
169|            case 'o':   base =  8  ;
170| pnum:
171|                /* Fetch a long or int sized argument off the
172|                 * stack as appropriate. If the fetched number
173|                 * is a base 10 int then mask off the top
174|                 * bits to prevent sign extension.
175|                 */
176|
177|                if( longf )
178|                    lnum = va_arg(args, long);
179|                else
180|                {
181|                    lnum = (long) va_arg(args, int);
182|
183|                    if( base == -10 )       /* Unsigned int */
184|                    {
185|                        base  = 10;
186|                        lnum &= INTMASK;
187|                    }
188|                    else if( base != 10 )   /* Nondecimal int */
189|                    {
190|                        lnum &= INTMASK;
191|                    }
192|                }
```

```
193|
194|                 if( lnum < 0L  &&  base == 10  && filchar == '0' )
195|                 {
196|                         /* Again, print a - to avoid "000-123."
197|                          * Only decimal numbers get - signs.
198|                          */
199|
200|                         (*out)( '-' ,o_param) ;
201|                         --fldwth ;
202|                         lnum = -lnum ;
203|                 }
204|
205|                 bp   = ltos( lnum, bp, base );
206|                 break;
207|             }
208|
209|             /*  Terminate the string if necessary and compute
210|              *  the string length (slen). Bp will point at the
211|              *  beginning of the output string.
212|              */
213|
214|             if (*format != 's')
215|             {
216|                 *bp  = '\0';
217|                 slen = bp - nbuf;
218|                 bp   = nbuf;
219|             }
220|             else
221|             {
222|                 slen = strlen(bp);
223|                 if( precision  &&  slen > precision )
224|                         slen = precision;
225|             }
226|
227|
228|             /*  Adjust fldwth to be the amount of padding we need
229|              *  to fill the buffer out to the specified field
230|              *  width. Then print leading padding (if we aren't
231|              *  left justifying), the buffer itself, and any
232|              *  required trailing padding (if we are left
233|              *  justifying.
234|              */
235|
236|             if( (fldwth -= slen) < 0 )
237|                 fldwth = 0;
238|
239|             if( center )
240|             {
241|                                 /* Use longf as counter */
242|                 longf = fldwth/2;
243|                 PAD( longf, filchar, out, o_param );
244|             }
245|             else if( !lftjust )
246|                 PAD( fldwth,   filchar, out, o_param );
247|
248|             while( --slen >= 0 )
249|                 (*out)(*bp++,o_param);
250|
251|             if( center )
252|             {
253|                 longf = fldwth - fldwth/2;
254|                 PAD( longf, filchar, out, o_param );
255|             }
256|             else if( lftjust)
257|                 PAD( fldwth, filchar, out, o_param );
258|
259|         }
260|     }
261| }
262|
263| /*---------------------------------------------------------*/
264|
265| #ifdef DEBUG
266|
267| #include <stdio.h>
268|
269| printm( fmt, ... )
270| char    *fmt;
271| {
272|         extern int  fputc();
273|         va_list args;
274|         va_start(args, fmt);
275|         idoprnt( fputc, stdout, fmt, args );
276| }
277|
278| /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
279|
280| main()
281| {
282|     printm("should see <0 1 2 3 4 5 6 7>: ");
283|     printm("           %d %x %o %ld %lx %lo %c %s\n",
284|                             0, 1, 2, 3L, 4L, 5L, '6', "7" );
285|
286|     printf("should see: hello world: "          );
287|     printm("%s %s %c", "hello", "world", '\n'   );
288|
289|     printm("should see <string>  : <%6.6s>\n",  "string NO NO" );
290|     printm("should see <   str>  : <%*.*s>\n",  6, 3, "string" );
291|     printm("should see <70000>   : <%ld>\n", 70000L     );
292|     printm("should see <fffff>   : <%lx>\n",0xfffffL);
293|     printm("should see <ffff>    : <%x>\n",     -1      );
294|     printm("should see <-1>      : <%ld>\n",    -1L     );
295|     printm("should see <x>       : <%c>\n",     'x'     );
296|     printm("should see <  x  >   : <%|5c>\n",   'x'     );
297|     printm("should see <a5>      : <%x>\n",     0xa5    );
298|     printm("should see <765>     : <%o>\n",     0765    );
299|     printm("should see <1010>    : <%b>\n",     0xa     );
300|     printm("should see <   123>  : <%6d>\n",    123     );
301|     printm("should see <   456>  : <%*d>\n",  6,456     );
302|     printm("should see <  -123>  : <%6d>\n",   -123     );
303|     printm("should see <123   >  : <%-6d>\n",   123     );
304|     printm("should see <-123  >  : <%-6d>\n",  -123     );
305|     printm("should see <-00123>  : <%06d>\n",  -123     );
306|     printm("should see <abcd:123>: <%p>\n",
307|                                                 (char far *)( 0xabcd0123L) );
308| }
309| #endif
```

End Listing One

Listing Two

```
1| /* LTOS.C        Convert long to string in indicated base.
2|  *               (C) 1988 Allen I. Holub. All rights reserved.
3|  */
4|
5| char    *ltos( n, buf, base )
6| unsigned long   n    ;
7| char           *buf ;
8| int            base ;
9| {
10|     /*  Convert long to string. Prints in hex, decimal, octal,
11|      *                                              or binary.
12|      *  "n"    is the number to be converted
13|      *  "buf"  is the output buffer.
14|      *  "base" is the base (16, 10, 8 or 2).
15|      *
16|      *  The output string is null terminated and a pointer to the
17|      *  null terminator is returned.
18|      *
19|      *  The number is put into an array one digit at a time as
20|      *  it's translated. The array is filled with the digits
21|      *  reversed (ie. the \0 goes in first, then the rightmost
22|      *  digit, etc.) and then is reversed in place before
23|      *  returning.
24|      *
25|      *  This routine is much like the unix() ltoa except that
26|      *  it returns a pointer to the end of the string.
27|      */
28|
29|     register char       *bp   = buf;
30|     register int        minus = 0;
31|     char               *endp;
32|
33|     if( base < 2 || base > 16 )
34|             return 0;
35|
36|     if( base == 10 && (long)n < 0 ) /* If the number is negative */
37|     {                               /* and we're in base 10, set */
38|         minus++ ;                   /* minus to true and make it */
39|         n = -( (long)n );           /* positive.                 */
40|     }
41|
42|     *bp = '\0' ;            /* Have to put the null in now */
43|                             /* because the array is being  */
44|                             /* filled in reverse order.    */
45|     do {
46|             *++bp = "0123456789abcdef" [ n % base ];
47|             n /= base;
48|
49|     } while( n );
50|
51|     if( minus )
52|             *++bp = '-';
53|
54|     for( endp = bp; bp > buf ;)      /* Reverse string in place */
55|     {
56|             minus  = *bp;           /* Use minus for temporary */
57|             *bp -- = *buf;           /* storage.                */
58|             *buf++ = minus;
59|     }
60|
61|     return endp;        /* Return pointer to terminating null   */
62| }
```

End Listings

TO THE MACS

Listing One *(Text begins on page 106.)*

```
-----------------------------------------------------------

Scouting Toolkit™
A HyperCard Project

Programmed and ©1988 by Stan Krute
All rights reserved

-----------------------------------------------------------

Description Of Objects

-----------------------------------------------------------

The Stack
   Description
      The stack's name is "Scouting Toolkit™"
      It contains 1 background
      It contains 1 card
      The size of the stack is 20K
   Script
      The stack has no script

-----------------------------------------------------------

Background 1
   Description
      The background has no name
      It's used by 1 card
   Script
      The background has no script

-----------------------------------------------------------

Card 1
   Description
      The card's name is "Scouting Toolkit"
      It contains 17 buttons
      It contains 2 fields
   Script
      ---------------------------- openCard -------------------------
      on openCard
```

```
    -- hide the menubar, and initialize a state variable for it
    global menubarState
    hide menubar
    put "hidden" into menubarState

    -- hide some windows
    hide message box
    hide tool window
    hide pattern window
  end openCard

  ---------- mouseUp ----------
  on mouseUp
    -- redraw the toolkit buttons, in case some are hidden
    repeat with buttonNumber = 2 to 13
      show button buttonNumber
    end repeat
  end mouseUp
```

Button 1

Description

The button's name is "Delete Scouting Toolkit"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30010
Its rectangle is 42 units wide, 42 units high
The button's located at card location (36,292)

Its text properties are: System Font 12, Height 16, Align center

Script

```
  ---------- mouseUp ----------
  on mouseUp
    -- make sure we don't mess up the supply card
    if short name of this card is not "Scouting Toolkit" then

      -- save and set the user level
      get userLevel
      put it into entryUserLevel
      set userLevel to 5

      -- show the watch cursor
      set cursor to 4

      -- make sure the kit buttons are showing
      show button "Scouting Toolkit Icon"
      click at loc of button "Scouting Toolkit Icon"

      -- set the tool
      choose button tool

      -- get rid of our toolkit
      click at loc of button "Home"
      doMenu "Clear Button"

      click at loc of button "Back"
      doMenu "Clear Button"

      click at loc of button "Next"
      doMenu "Clear Button"

      click at loc of button "Previous"
      doMenu "Clear Button"

      click at loc of button "Toggle Pattern Window"
      doMenu "Clear Button"

      click at loc of button "Toggle Tool Window"
      doMenu "Clear Button"

      click at loc of button "Toggle Message Box"
      doMenu "Clear Button"

      click at loc of button "Toggle Menubar"
      doMenu "Clear Button"

      choose field tool
      show card field "Copyright"
      click at loc of card field "Copyright"
      doMenu "Clear Field"
      choose button tool

      click at loc of button "Scouting Toolkit About"
      doMenu "Clear Button"

      click at loc of button "Scouting Toolkit™"
      doMenu "Clear Button"

      click at loc of button "Scouting Toolkit Window"
      doMenu "Clear Button"

      show button "Scouting Toolkit Icon"
      click at loc of button "Scouting Toolkit Icon"
      doMenu "Clear Button"

      -- snake swallows tail
      click at loc of button "Delete Scouting Toolkit"
      doMenu "Clear Button"

      -- set the tool
      choose browse tool

      -- restore the cursor
      set cursor to 0
      -- restore the user level
      set userLevel to entryUserLevel

    else
      -- we're on the supply card, so it would be bad form to delete

      -- send a signal
      show card field "No Delete Here"

      -- wait a moment
      wait 1 seconds

      -- hide the signal
      hide card field "No Delete Here"
    end if
  end mouseUp
```

Button 2

Description

The button's name is "Scouting Toolkit Window"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has no icon
Its rectangle is 300 units wide, 182 units high
The button's located at card location (197,144)
Its text properties are: System Font 12, Height 16, Align center

Script

```
  ---------- mouseUp ----------
  on mouseUp
    -- make a noise
    play "boing" "c"

    -- close the kit
    hideScoutingToolkit

    -- show the trigger
    show button "Scouting Toolkit Icon"
  end mouseUp


  ---------- hideScoutingToolkit ----------
  on hideScoutingToolkit
    -- hide the kit's buttons
    hide button "Home"
    hide button "Back"
    hide button "Next"
    hide button "Previous"
    hide button "Toggle Pattern Window"
    hide button "Toggle Tool Window"
    hide button "Toggle Message Box"
    hide button "Toggle Menubar"
    hide button "Scouting Toolkit About"
    hide button "Scouting Toolkit™"
    hide button "Scouting Toolkit Window"
  end hideScoutingToolkit
```

Button 3

Description

The button's name is "Scouting Toolkit™"
Its Show name property is true
Its Auto hilite property is true
Its style is opaque
It has no icon
Its rectangle is 120 units wide, 20 units high
The button's located at card location (296,160)
Its text properties are: System Font 12, Height 16, Align center

Script

```
  ---------- mouseUp ----------
  on mouseUp
    -- pass the message through
    send mouseUp to button "Scouting Toolkit Window"
  end mouseUp
```

Button 4

Description

The button's name is "Scouting Toolkit About"
Its Show name property is false
Its Auto hilite property is true
Its style is transparent
It has an icon, ID# 30011
Its rectangle is 24 units wide, 22 units high
The button's located at card location (466,150)
Its text properties are: System Font 12, Height 16, Align center

Script

```
  ---------- mouseUp ----------
  on mouseUp
    -- show the copyright notice
    show card field "Copyright"

    -- a little sound effect
```

```
    play "harpsichord" "c e g"

    -- wait for a mouse click
    wait until the mouseClick

    -- a little sound effect
    play "harpsichord" "g e c"

    -- hide the copyright notice
    hide card field "Copyright"
  end mouseUp
```

Button 5

Description

```
The button's name is "Toggle Menubar"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30009
Its rectangle is 42 units wide, 42 units high
The button's located at card location (225,193)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- no built-in check for visibility, so…
  global menubarState

  -- toggle the menubar's visibility
  if menubarState is "hidden" then
    show menubar
    put "showing" into menubarState
  else
    hide menubar
    put "hidden" into menubarState
  end if
end mouseUp
```

Button 6

Description

```
The button's name is "Toggle Message Box"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30008
Its rectangle is 42 units wide, 42 units high
The button's located at card location (292,193)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- toggle the message window's visibility
  if the visible of message box is true then
    hide the message box
  else
    show the message box
  end if
end mouseUp
```

Button 7

Description

```
The button's name is "Toggle Tool Window"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30007
Its rectangle is 42 units wide, 42 units high
The button's located at card location (359,193)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- toggle the tool palette window's visibility
  if the visible of tool window is true then
    hide tool window
  else
    show tool window
  end if
end mouseUp
```

Button 8

Description

```
The button's name is "Toggle Pattern Window"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30006
Its rectangle is 42 units wide, 42 units high
The button's located at card location (426,193)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- toggle the pattern palette's window's visibility
  if the visible of pattern window is true then
    hide pattern window
  else
    show pattern window
  end if
end mouseUp
```

Button 9

Description

```
The button's name is "Previous"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30001
Its rectangle is 42 units wide, 42 units high
The button's located at card location (225,259)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- move to the previous card
  doMenu "Prev"
end mouseUp
```

Button 10

Description

```
The button's name is "Next"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30002
Its rectangle is 42 units wide, 42 units high
The button's located at card location (292,259)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- move to the next card
  doMenu "Next"
end mouseUp
```

Button 11

Description

```
The button's name is "Back"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30003
Its rectangle is 42 units wide, 42 units high
The button's located at card location (359,259)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- go to the last card visited
  doMenu "Back"
end mouseUp
```

Button 12

Description

```
The button's name is "Home"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30005
Its rectangle is 42 units wide, 42 units high
The button's located at card location (426,259)
Its text properties are: System Font 12, Height 16, Align center
```

Script

```
---------- mouseUp ----------
on mouseUp
  -- go home
  visual effect iris close
  doMenu "Home"
end mouseUp
```

Button 13

Description

```
The button's name is "Scouting Toolkit Icon"
Its Show name property is false
Its Auto hilite property is true
Its style is shadow
It has an icon, ID# 30004
Its rectangle is 42 units wide, 42 units high
The button's located at card location (178,83)
```

```
    Its text properties are: System Font 12, Height 16, Align center
  Script
    ------------------------------ mouseUp --------------------------
    on mouseUp
      -- make a noise
      play "boing" "e"

      -- hide this button
      hide button "Scouting Toolkit Icon"

      -- show the toolkit
      showScoutingToolkit
    end mouseUp
    ---------------------- showScoutingToolkit --------------------
    on showScoutingToolkit
      -- show the toolkit's buttons
      show button "Toggle Menubar"
      show button "Scouting Toolkit Window"
      show button "Scouting Toolkit About"
      show button "Scouting Toolkit™"
      show button "Toggle Message Box"
      show button "Toggle Tool Window"
      show button "Toggle Pattern Window"
      show button "Previous"
      show button "Next"
      show button "Back"
      show button "Home"
    end showScoutingToolkit
-------------------------------------------------------------
Button 14
  Description
    The button's name is "Duplicate Scouting Toolkit"
    Its Show name property is false
    Its Auto hilite property is true
    Its style is shadow
    It has an icon, ID# 30000
    Its rectangle is 42 units wide, 42 units high
    The button's located at card location (460,12)
    Its text properties are: System Font 12, Height 16, Align center
  Script
    ------------------------------ newButton ------------------------
    on newButton
      -- make sure we don't mess up the supply card
      if short name of this card is not "Scouting Toolkit" then

    -- save and set the user level
    get userLevel
    put it into entryUserLevel
    set userLevel to 5

    -- set the cursor to the watch
    set cursor to 4

    -- set the tool
    choose button tool

    -- for all the buttons in the set
    repeat with buttonNumber = 1 to 13

      -- remember where we are (the target card)
      push card

      -- hide screen commotion
      set lockScreen to true

      -- go to the supply card
      go to card "Scouting Toolkit" of stack "Scouting Toolkit™"

      -- select the button
      click at loc of button buttonNumber
      -- copy the button
      doMenu "Copy Button"

      -- return to the target card
      pop card

      -- show screen commotion
      set lockScreen to false

      -- paste the button
      doMenu "Paste Button"

    end repeat

    -- copy the copyright notice
    copyCopyrightField

    -- get rid of this duplication button
    choose button tool
    click at loc of button "Duplicate Scouting Toolkit"
    doMenu "Cut Button"

    -- move two buttons
    drag from loc of button "Scouting Toolkit Icon" to loc of button "Back"
    hide button "Scouting Toolkit Icon"
    drag from loc of button "Delete Scouting Toolkit" to loc of button "Home"

    -- hide the toolkit
    choose browse tool
    click at loc of button "Scouting Toolkit Window"

    -- restore the user level
    set userLevel to entryUserLevel
  end if
end newButton
------------------------ copyCopyrightField ----------------------
on copyCopyrightField
  -- save our location
  push card

  -- hide screen commotion
  set lockScreen to true

  -- go to the supply card
  go to card "Scouting Toolkit" of stack "Scouting Toolkit™"

  -- set tool
  choose field tool

  -- select the field
  show card field "Copyright"
  click at loc of card field "Copyright"

  -- copy the button
  doMenu "Copy Field"

  -- hide the field
  hide card field "Copyright"
  --
  -- and get the field's contents
  get card field "Copyright"

  -- return to the target card
  pop card

  -- show screen commotion
  set lockScreen to false

  -- paste the field, fill it up, and hide it
  doMenu "Paste Field"
  put It into card field "Copyright"
  hide card field "Copyright"
end copyCopyrightField
------------------------------ mouseUp -------------------------------
on mouseUp
  -- make sure we're at the supply card
  if short name of this card is "Scouting Toolkit" then

    -- show the watch cursor
    set cursor to 4
        -- copy the button
        choose button tool
        click at loc of button "Duplicate Scouting Toolkit"
        doMenu "Copy Button"

        -- restore tool
        choose browse tool
      end if
    end mouseUp
-------------------------------------------------------------
Button 15
  Description
    The button's name is "Copyright"
    Its Show name property is false
    Its Auto hilite property is true
    Its style is transparent
    It has an icon, ID# 30000
    Its rectangle is 70 units wide, 13 units high
    The button's located at card location (0,52)
    Its text properties are: System Font 12, Height 16, Align center
  Script
    ------------------------------ mouseUp -------------------------
    on mouseUp
      -- pass message on to the toolkit's About button
      send mouseUp to button "Scouting Toolkit About"
    end mouseUp
-------------------------------------------------------------
Card Field 1
  Description
    The field's name is "Copyright"
    It's hiddden
    Its Lock text property is true
    Its Show lines property is false
    Its Wide margins property is false
    Its style is rectangle
    Its rectangle is 286 units wide, 129 units high
    The field's located at card location (204,190)
    Its text properties are: Geneva 12, Height 16, Align center
  Contents
    "©1987 by Stan Krute's Camp Creek Institute.
    All rights reserved.
    Call or write for details:
    18617 Camp Creek Road
    Hornbrook, California96044
    [916] 475-3428"
  Script
    The field has no script
-------------------------------------------------------------
Card Field 2
  Description
    The field's name is "No Delete Here"
    It's hiddden
    Its Lock text property is true
    Its Show lines property is false
    Its Wide margins property is false
    Its style is opaque
    Its rectangle is 39 units wide, 39 units high
    The field's located at card location (37,293)
    Its text properties are: Geneva 9, Height 12, Align center
  Contents
    "Not Here,
    Though."
  Script
    The field has no script
```

End Listing

STRUCTURED PROGRAMMING

Listing One *(Text begins on page 118.)*

```
DEFINITION MODULE LineDwg;
  EXPORT QUALIFIED
    width, height, CharWidth, CharHeight, PaintMode, Px, Py,
    mode, dot, line, paint, copyArea, clear, Write, WriteString;

  TYPE PaintMode = (replace, add, invert, erase);

  VAR Px, Py     : INTEGER; (* Current coordinates of drawing pen *)
      mode       : PaintMode;  (* Current mode for paint and copy *)
      width      : INTEGER;   (* Width of picture area, read-only *)
      height     : INTEGER;  (* Height of picture area, read-only *)
      CharWidth  : INTEGER;              (* Width of a character *)
      CharHeight : INTEGER;             (* Height of a character *)

  PROCEDURE dot (c : CARDINAL; x, y : INTEGER);
        (* Place a dot at coordinate x, y in color c *)

  PROCEDURE line (d, n : CARDINAL);
        (* Draw a line of length n in direction d
           (angle = 45 * d degrees) *)

  PROCEDURE paint (c : CARDINAL; x, y, w, h : INTEGER);
        (* Paint the rectangular area at x, y of width w and
           height h in color c, where 0 = white, 1 = light gray,
           2 = dark gray, 3 = black *)
        (* Note: This proc is also called 'area' by Wirth *)

  PROCEDURE copyArea (sx, sy, dx, dy, dw, dh : INTEGER);
        (* Copy rectangular area at sx, sy into rectangle at dx, dy
           of width dw and height dh *)

  PROCEDURE clear;         (* Clear the screen *)
        (* Note: Also places display in EGA 640 x 350 color mode *)

  PROCEDURE Write (ch : CHAR);   (* Write ch at pen's position *)

  PROCEDURE WriteString (s : ARRAY OF CHAR);

END LineDwg.
```

End Listing One

Listing Two

```
IMPLEMENTATION MODULE LineDwg;

(* Implements LineDrawing module defined by N. Wirth in *)
(*   "Programming in Modula-2." This module assumes EGA *)
(*   monitor in 640 x 350 color graphics mode.          *)
(* Uses virtual coordinates, where origin is at lower   *)
(*   left corner of screen area, size is 800 x 600.     *)
(* Adapted by K. Porter for DDJ, April 1988 issue       *)
(* ---------------------------------------------------- *)

  FROM SYSTEM IMPORT REGISTERS, INT;
  FROM Strings IMPORT Length;

  CONST  VW  = 800.0;         (* virtual width of screen *)
         VH  = 600.0;                (* virtual height *)
         RW  = 640.0;  (* real device width, EGA screen *)
         RH  = 350.0;                (* device height *)
         EGA = 16;          (* EGA 640 x 350 color mode *)

  (* Variables local to this module *)
  VAR    reg    : REGISTERS;
         xf, yf : REAL;
         color  : INTEGER;

  (* --------------LOCAL PROCEDURES ------------------------- *)

  PROCEDURE writePixel (c, x, y : INTEGER);
                    (* write pixel of color c at device x, y *)

  BEGIN
    reg.AH := 12;
    CASE c OF           (* map color indicator to EGA palette *)
      0 : reg.AL := 15|                              (* white *)
      1 : reg.AL :=  7|                         (* light gray *)
      2 : reg.AL :=  8|                          (* dark gray *)
      3 : reg.AL :=  0                               (* black *)
    END;  (* of CASE *)
    reg.BX := 0;
    reg.CX := x;
    reg.DX := y;
    INT (16, reg);
    color := c;                        (* set prevailing color *)
  END writePixel;
  (* ------------------------ *)

  PROCEDURE devX (x : INTEGER) : INTEGER;
                                  (* translate x to device x *)

  BEGIN
    RETURN TRUNC (xf * FLOAT (x));
  END devX;
  (* ------------------------ *)

  PROCEDURE devY (y : INTEGER) : INTEGER;
                                  (* translate y to device y *)

  BEGIN
    RETURN TRUNC (RH - (yf * FLOAT (y)));
  END devY;


(* ------------------ VISIBLE PROCEDURES -------------------- *)

  PROCEDURE dot (c : CARDINAL; x, y : INTEGER);

        (* Place a dot of color c at coordinate x, y *)

  BEGIN
    writePixel (c, devX (x), devY (y));
  END dot;
  (* ------------------------ *)

  PROCEDURE line (d, n : CARDINAL);

        (* Draw a line of length n in direction d
          (angle = 45 * d degrees) *)

  VAR  xdir, ydir : INTEGER;     (* x and y directions given d *)
       distance   : CARDINAL;

  BEGIN
    CASE d OF
      0 : xdir :=  1; ydir :=  0|                       (* right *)
      1 : xdir :=  1; ydir :=  1|
      2 : xdir :=  0; ydir :=  1|                          (* up *)
      3 : xdir := -1; ydir :=  1|
      4 : xdir := -1; ydir :=  0|                        (* left *)
      5 : xdir := -1; ydir := -1|
      6 : xdir :=  0; ydir := -1|                        (* down *)
      7 : xdir :=  1; ydir := -1
    END;     (* of CASE *)
    FOR distance := 1 TO n DO
      Px := Px + xdir;                        (* advance the pen *)
      Py := Py + ydir;
      dot (color, Px, Py);          (* draw in prevailing color *)
    END;
  END line;
  (* ------------------------ *)
  PROCEDURE paint (c : CARDINAL; x, y, w, h : INTEGER);

   (* Paint the rectangular area at x, y of width w and
            height h in color c, where 0 = white,
            1 = light gray, 2 = dark gray, 3 = black *)

   VAR  cy, prevY, dy : INTEGER;

   BEGIN
     prevY := 0;
     color := c;                      (* set new prevailing color *)
     FOR cy := y TO y+h DO
       dy := devY (cy);                  (* get current device y *)
       IF dy <> prevY THEN             (* if new scan line, draw *)
         Px := x; Py := cy;
         line (0, w);
         prevY := dy;          (* remember where last line drawn *)
       END
     END
   END paint;
   (* ------------------------ *)

   PROCEDURE copyArea (sx, sy, dx, dy, dw, dh : INTEGER);

         (* Copy rectangular area at sx, sy into rectangle
            at dx, dy of width dw and height dh *)

   VAR  c, x, y, ix, iy, nx, ny, tx, ty : INTEGER;

   BEGIN
     ix := devX (sx);    iy := devY (sy); (* source dev coords *)
     nx := devX (sx+dw); ny := devY (sy+dh);  (* ending coords *)
     tx := devX (dx);    ty := devY (dy);     (* target coords *)
     FOR y := ny TO iy DO                  (* go top to bottom *)
       FOR x := ix TO nx DO
         reg.AH := 13;                            (* read pixel *)
         reg.BX := 0;
```

(continued on next page)

Listing Two

(Listing continued, text begins on page 118.)

```
      reg.CX := x;
      reg.DX := y;
      INT (16, reg);              (* get pixel color into al *)
      reg.AH := 12;                  (* write pixel to dest *)
      reg.CX := tx + x - ix;
      reg.DX := ty + y - iy;
      INT (16, reg);
    END
  END
END copyArea;
(* ------------------------ *)

PROCEDURE clear;                            (* Clear the screen *)
            (* Also places display into EGA graphics mode *)

BEGIN
  reg.AH := 0;
  reg.AL := EGA;                              (* EGA 640 x 350 *)
  INT (16, reg);
  color := 0;                 (* reset default color to white *)
END clear;
(* ------------------------ *)

PROCEDURE Write (ch : CHAR);  (* Write ch at pen's position *)

VAR  cc, cr : INTEGER;                    (* Char col and row *)

BEGIN
  cc := devX (Px) DIV  8;         (* Derive char pos from pen *)
  cr := devY (Py) DIV 14;
  reg.AH := 2;                    (* Set text cursor position *)
  reg.BX := 0;
  reg.DX := (cr * 256) + cc;
  INT (16, reg);
  reg.AX := 2560 + ORD (ch);       (* Write char via ROM BIOS *)
  reg.BX := 7;                        (* Light gray text only *)
  reg.CX := 1;
  INT (16, reg);
  Px := Px + CharWidth;     (* advance by char virtual width *)
END Write;
(* ------------------------ *)

PROCEDURE WriteString (s : ARRAY OF CHAR);

VAR  i : CARDINAL;

BEGIN
    FOR i := 0 TO Length (s) DO
      Write (s[i]);
    END
  END WriteString;

  (* ---------------- INITIALIZATION ------------------------ *)

BEGIN
  Px         := TRUNC (VW / 2.0);     (* Virtual screen center *)
  Py         := TRUNC (VH / 2.0);
  mode       := replace;
  width      := TRUNC (VW);             (* Virtual screen size *)
  height     := TRUNC (VH);
  CharWidth  := 10;              (* Char sizes in virtual units *)
  CharHeight := 24;
  xf         := RW / VW;                (* x translation factor *)
  yf         := RH / VH;                (* y translation factor *)
  color      := 0;                    (* white is default color *)
END LineDwg.
```

End Listing Two

Listing Three

```
MODULE Sierpin;

(* Based on the Sierpinski recursive model in N. Wirth's book *)
(*   "Programming in Modula-2."                                *)
(* Wirth's program has been modified slightly to accommodate  *)
(*   EGA graphics mode on the IBM PC.                          *)
(* Written by K. Porter for DDJ, April 1988 issue.            *)
(* ---------------------------------------------------------- *)

FROM InOut   IMPORT Read;
FROM LineDwg IMPORT width, height, Px, Py, clear, line;
FROM SYSTEM  IMPORT REGISTERS, INT;
```

(continued on page 92)

STRUCTURED PROGRAMMING

Listing Three

(Listing continued, text begins on page 118.)

```
CONST SquareSize = 512;

VAR  i, h, x0, y0 : CARDINAL;
     ch           : CHAR;
     reg          : REGISTERS;

PROCEDURE A (k : CARDINAL);
BEGIN
  IF k > 0 THEN
    A (k-1);   line (7, h);   B (k-1);   line (0, 2*h);
    D (k-1);   line (1, h);   A (k-1)
  END
END A;
(* ------------------------- *)

PROCEDURE B (k : CARDINAL);
BEGIN
  IF k > 0 THEN
    B (k-1);   line (5, h);   C (k-1);   line (6, 2*h);
    A (k-1);   line (7, h);   B (k-1)
  END
END B;
(* ------------------------- *)

PROCEDURE C (k : CARDINAL);
BEGIN
  IF k > 0 THEN
    C (k-1);   line (3, h);   D (k-1);   line (4, 2*h);
    B (k-1);   line (5, h);   C (k-1)
  END
END C;
(* ------------------------- *)

PROCEDURE D (k : CARDINAL);
BEGIN
  IF k > 0 THEN
    D (k-1);   line (1, h);   A (k-1);   line (2, 2*h);
    C (k-1);   line (3, h);   D (k-1)
  END
END D;
(* ------------------------- *)

BEGIN
  clear;
  i := 0;
  h := SquareSize DIV 4;
  x0 := CARDINAL (width) DIV 2;
  y0 := CARDINAL (height) DIV 2 + h;
  REPEAT
    i  := i + 1;
    x0 := x0 - h;
    h  := h DIV 2;
    y0 := y0 + h;
    Px := x0;
    Py := y0;
    A (i);   line (7, h);   B (i);   line (5, h);
    C (i);   line (3, h);   D (i);   line (1, h);
  UNTIL (i = 4);
  Read (ch);                                (* hold for keypress *)
  reg.AH := 0;
  reg.AL := 3;              (* restore 80 x 25 color text mode *)
  INT (16, reg);
END Sierpin.
```

End Listing Three

Listing Four

```
  MODULE MathPlot;

  (* Graphs the function y = cos x + sin 2x using LineDwg *)
  (* K. Porter, DDJ, April 88                             *)

  FROM MathLib0 IMPORT sin, cos, real, entier;
  FROM LineDwg IMPORT line, dot, clear, WriteString, Px, Py;
  FROM InOut IMPORT Read;
  FROM SYSTEM IMPORT REGISTERS, INT;

  VAR  x, sx, cx, y : REAL;
       xc           : INTEGER;
       ch           : CHAR;
       reg          : REGISTERS;

  PROCEDURE yc (y : REAL) : INTEGER;
  BEGIN
    RETURN entier (y * 100.0) + 300;
  END yc;
```

```
BEGIN
  clear;
  Px :=   0; Py := 300; line (0, 800);          (* x axis *)
  Px :=   0; Py := yc (1.0);                    (* grids *)
  WriteString ("  1");
  dot (2, Px, Py);                          (* set dark gray *)
  line (0, 750);
  Px := 0; Py := yc (-1.0);
  WriteString (" -1"); line (0, 750);
  FOR xc := 125 TO 799 BY 126 DO       (* vertical grids *)
    Px := xc; Py := yc (2.0); line (6, 400);
  END;
  Px := 310; Py := 599;
  WriteString ("y = cos x + sin 2x ");

  FOR xc := 1 TO 799 DO
    x  := real (xc) / 80.0;              (* x to radians *)
    sx := sin (2.0 * x);
    dot (2, xc, yc (sx));      (* plot sine in dark gray *)
    cx := cos (x);
    dot (1, xc, yc (cx));          (* cos in light gray *)
    y  := sx + cx;
    dot (0, xc, yc (y));       (* plot function in white *)
  END;
  Read (ch);
  reg.AH := 0; reg.AL := 3; INT (16, reg)  (* text mode *)
END MathPlot.
```

End Listing Four

Listing Five

```
MODULE Spiral;

(* Draws a spiral, then copies part of it to a window *)

FROM SYSTEM IMPORT REGISTERS, INT;
FROM LineDwg IMPORT Px, Py, line, copyArea, WriteString, clear;
FROM InOut IMPORT Read;

VAR  reg  : REGISTERS;
     ch   : CHAR;
     n, d : INTEGER;

BEGIN
  clear;
  Px := 250;
  FOR n := 0 TO 24 DO                          (* Draw the spiral *)
    d := n MOD 8;
    line (d, n * 5);
  END;
  Px := 548; Py := 198;              (* Outline the copy window *)
  line (0, 204);
  line (2, 204);
  line (4, 204);
  line (6, 204);
  Px := 630; Py := 440;
  WriteString ("Copy ");
  copyArea (200, 200, 550, 200, 200, 200);     (* Copy portion *)
  Read (ch);                                  (* Wait for keypress *)
  reg.AH := 0; reg.AL := 3; INT (16, reg);
END Spiral.
```

End Listings

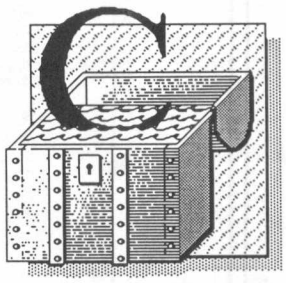
THE FORTH COLUMN

Listing One *(Text begins on page 130.)*

```
SCR# 0

( Corrected source screens)     The following four screens contain
corrected source for the operators  GETDATA PUTDATA and PUTTEXT.
The original uncorrected versions are in the DDJ Forth column, Feb
1988.

Screen 5 defines GETDATA using
BEGIN ... DUP IF  DROP ... THEN WHILE ... REPEAT
instead of the preferred
BEGIN ... WHILE ... WHILE ... REPEAT THEN

SCR# 1

\ Read BLOCKed file as data file

: GETDATA ( a n - n2)
\ reads n bytes of data from input file into address, n < 64K
\ Returns n2 bytes not read ( ie beyond end of file ).
    ( calculate # of bytes to move < 64K : )  POSITION 2@
    BEGIN  2 PICK ( n ) DUP
      IF ( n ) >R  2DUP 1K UM/MOD  SWAP DROP  1+  1K UM*
         CAPACITY 2@ DMIN  2OVER D- 0= NOT OR  R> UMIN
      THEN  ?DUP
    WHILE  >R  2DUP 1K UM/MOD  BLOCK +  4 PICK R@ CMOVE
     R@ 0 D+  2SWAP  R> /STRING  2SWAP
    REPEAT  POSITION 2!  SWAP DROP ;

SCR# 2


\ Write BLOCKed file as data file

: PUTDATA ( a n)
\ writes n bytes of data to output file from address, n < 64K
   ( extend the file as needed : )
   DUP 0  POSITION 2@ D+  CAPACITY 2@  2OVER D-  DUP 0<
   IF  2DUP DABS EXTEND  2OVER CAPACITY 2!  THEN  2DROP 2DROP
   ( calculate # of bytes to move < 64K : )  POSITION 2@
   BEGIN  2 PICK ( n ) DUP
      IF ( n ) >R  2DUP 1K UM/MOD  SWAP DROP  1+  1K UM*
         CAPACITY 2@ DMIN  2OVER D- 0= NOT OR  R> UMIN
      THEN  ?DUP
   WHILE  >R  2DUP 1K UM/MOD  BLOCK +  4 PICK SWAP R@ CMOVE
      R@ 0 D+  2SWAP  R> /STRING  2SWAP   UPDATE
   REPEAT  POSITION 2!  2DROP ;

SCR# 3


 0 \ Read text file with #EOF
 1
 2 : GETTEXT ( a n - n2 f)   POSITION 2@   0 ( f ) >R
 3 \ reads n bytes of text from input file into address, n < 64K
 4 \ Returns n2 bytes not read ( ie end-of-line or beyond file)
 5 \ Returns true if #EOL terminates line; false otherwise.
 6    BEGIN  2DUP CAPACITY 2@ D<  3 PICK ( n ) AND
 7    WHILE  2DUP 1K UM/MOD  BLOCK + C@ ( get ch)
 8         DUP  #EOL =    DUP R> OR >R
 9         OVER #EOF = OR NOT  AND ?DUP
10    WHILE  >R  1 0 D+  2SWAP  R> 2 PICK C!  1 /STRING  2SWAP
11    REPEAT THEN   2DUP CAPACITY 2@ D<
12    IF  2DUP 1K UM/MOD  BLOCK + C@  #EOL = DUP D-  THEN
13    2DUP CAPACITY 2@ D<
14    IF  2DUP 1K UM/MOD  BLOCK + C@  #LF  = DUP D-  THEN
15    POSITION 2!  SWAP DROP  R> ;


SCR# 4

 0 \ Read text file without #EOF
 1
 2 : GETTEXT ( a n - n2 f)   POSITION 2@   0 ( f ) >R
 3 \ reads n bytes of text from input file into address, n < 64K
 4 \ Returns n2 bytes not read ( ie end-of-line or beyond file)
 5 \ Returns true if #EOL terminates line; false otherwise.
 6    BEGIN  2DUP CAPACITY 2@ D<  3 PICK ( n ) AND
 7    WHILE  2DUP 1K UM/MOD  BLOCK + C@ ( get ch)
 8       DUP  #EOL =    DUP R> OR >R
 9                      NOT  AND ?DUP
10    WHILE  >R  1 0 D+  2SWAP  R> 2 PICK C!  1 /STRING  2SWAP
11    REPEAT THEN   2DUP CAPACITY 2@ D<
12    IF  2DUP 1K UM/MOD  BLOCK + C@  #EOL = DUP D-  THEN
13    2DUP CAPACITY 2@ D<
14    IF  2DUP 1K UM/MOD  BLOCK + C@  #LF  = DUP D-  THEN
15    POSITION 2!  SWAP DROP  R> ;


SCR# 5
 0 \ Read text file with #EOF
 1
 2 : GETTEXT ( a n - n2 f)   POSITION 2@   0 ( f ) >R
 3 \ reads n bytes of text from input file into address, n < 64K
 4 \ Returns n2 bytes not read ( ie end-of-line or beyond file)
 5 \ Returns true if #EOL terminates line; false otherwise.
 6    BEGIN  2DUP CAPACITY 2@ D<  3 PICK ( n ) AND
 7    DUP IF  DROP  2DUP 1K UM/MOD  BLOCK + C@ ( get ch)
 8            DUP  #EOL =    DUP R> OR >R
 9            OVER #EOF = OR NOT  AND ?DUP  THEN
10    WHILE  >R  1 0 D+  2SWAP  R> 2 PICK C!  1 /STRING  2SWAP
11    REPEAT THEN   2DUP CAPACITY 2@ D<
12    IF  2DUP 1K UM/MOD  BLOCK + C@  #EOL = DUP D-  THEN
13    2DUP CAPACITY 2@ D<
14    IF  2DUP 1K UM/MOD  BLOCK + C@  #LF  = DUP D-  THEN
15    POSITION 2!  SWAP DROP  R> ;
```

End Listing

Formatted Print Functions: The Innards



A few months ago (August 1987), I presented the basic theory behind writing a subroutine with a variable number of arguments. This month I'm going to apply the theory by presenting a version of *printf( )*. "Why rewrite *printf( )*," you may well ask, "when you already have a version in the library?" There are two main reasons: size and features.

My version is an integer-only version of the subroutine. Because the normal *printf( )* can handle floating-point conversions, the linker will call the entire floating-point library into your program, even if it doesn't use floating point. The second size-related problem is implementation dependent. Microsoft's *printf( )* uses up an inordinate amount of stack space (which causes several problems in some applications, such as programs that use the multitasking kernel presented in this column in December and January). I suspect that it builds the entire output string in memory before sending it to the output device. My own version uses a different approach, which I'll discuss later.

The other issue is features. I wanted a %*b* (for binary) conversion that would write out a number in base 2, and I wanted some mechanism for centering an object in a field (as compared the default right justification or the left justification available with a minus-sign modifier). The various conversions supported by my *printf( )* are detailed in Table 1, page 100.

by Allen Holub

The %*p* conversion prints a 32-bit *far* pointer in a *SEG:OFFSET* form (use %*x* to print a 16-bit *near* pointer). To use it in a small-model program, you'll have to use a cast:

```
printf("%p", (void far *)ptr );
```

Note that it's impossible to get zero fill in the offset part of a %*p* conversion. That is 1234:0001 is always printed as 1234:1. You can use %*0p* to pad out the segment portion, however.

Implementation

Printf( ), *fprintf( )*, and *sprintf( )* all use a single workhorse function to do the work. This function, called *idoprnt( )*, is called with four arguments: a pointer to an output function, an argument to pass to that function in addition to the output character, the format-string pointer, and a pointer to the position on the stack of the rest of the arguments. Examples 1 and 2, below, show how it's used.

Figure 1, page 104, shows a picture of the stack after a call to:

```
sprintf( buf, "%d %lx", 1, (long)2 );
```

(I've chosen the *sprintf( )* call because it's the most complicated.) Arguments are pushed on the stack in reverse order, so the rightmost two arguments are pushed first: a *long int* holding the number 2 and a normal *int* holding the number 1. A pointer to the format string is pushed next, followed by a pointer to the output buffer. *Sprintf( )* calls *idoprnt( )*, pushing a pointer to the first argument, a duplicate of the format-string pointer (if you don't understand this, go back and read the August C Chest), the address of the buffer pointer and a pointer to the output routine *putstr( )*, which will be called to output every character. *Putstr( )* is passed the same pointer to the buffer pointer that was passed to *idoprnt( )*, along with the character to output.

So, *putstr( )* outputs a character by putting it into ***p*. A glance at Figure 1 shows you that two levels of dereferencing gets you to a character in the buffer itself. The routine then increments **p*—that is, it increments the *buf* variable that's in *sprintf( )*'s stack frame.

Idoprnt( ) is presented in Listing One, page 68. I've done a few questionable things here—at least from the point of view of structured programming—primarily for speed reasons. If you run the profiler on most C programs, you'll find that a large

```
#include <stdio.h>
#include <stdarg.h>

printf( fmt, ... )
char    *fmt;
{
    extern int  fputc();
    va_list     args;

    va_start(args, fmt);
    idoprnt( fputc, stdout, fmt, args );
}

fprintf( stream, fmt, ... )
FILE    *stream;
char    *fmt;
{
    extern int  fputc();
    va_list     args;

    va_start(args, fmt);
    idoprnt( fputc, stream, fmt, args );
}
```

Example 1: Printf *and* fprintf

```
#include <stdarg.h>

putstr(c, p)
int     c;
char    **p;
{
    **p = c ;
    (*p)++  ;
}

int sprintf( buf, fmt, ... )
FILE    *stream;
char    *fmt;
{
    extern int  fputc();
    va_list     args;

    char *start = buf;
    va_start(args, fmt);
    idoprnt( fputc, &buf, fmt, args );
    *buf = '\e0' ;
    return buf - start;
}
```

Example 2: Sprintf

(continued from page 98)

percentage of the program's execution time is spent in *printf( )*. Consequently, it's worthwhile to speed things up a little, even if the code suffers a bit as a consequence. Also note that I've used the ANSI (not Unix) variable-argument conventions, all defined in <stdarg.h>, included on line 1.

The <dos.h> file included on line 2 is supplied by Microsoft and contains *#defines* for the *FP_OFF* and *FP_SEG* macros, which are used to extract the offset and segment portions of a *far* pointer. There really isn't a portable way to do this, but one possibility is:

```
#define FP_OFF(fp) ((unsigned)(fp))
#define FP_SEG(fp) ((unsigned)
        ((unsigned long)(fp) >> 16))
```

Note that I'm counting on truncating a 32-bit pointer to 16 bits in the cast to *unsigned*. You have to cast the pointer to a *long int* in order to shift it.

There are three macros of interest on lines 57–68 of Listing One. *PAD* outputs *fw* characters (*fc*), using the indicated output-function pointer (*out*) and passing that function an additional argument (*op*). To send five spaces to *stdout*, use:

```
int fp = 5;
PAD( fp, ' ', fputc, stdout );
```

The first argument must be a variable reference (not a constant). Note that an undesirable side effect of *PAD* sets *fp* to 0 when it terminates.

TOINT(p,x) converts the integer represented by the string *p* to an *int* and puts the result into *x*. It modifies *p* to point past the rightmost digit. *INTMASK* is a portable way to mask off the bottom *int*-size number of bits in a *long*. That is, given a 16-bit *int* and a 32-bit *long*, I want to mask out the bottom 16 bits. I can't say:

```
long x;
x &= 0xffff ;
```

because *0xffff* is treated by the compiler as an *int*. Moreover, it's negative. The *&=* will cause an implicit type conversion from *int* to *long*, performing a sign extension as part of the conversion (*0xffff* will be converted to *0xffffffff*), and the *AND* operation will have no effect. To get around this problem, you have to cast the *0xffff* to *unsigned* to defeat the sign extension:

```
x &= (unsigned)0xffff ;
```

The next problem is that *0xffff* assumes that an *int* is 16 bits. It's better to say:

```
void printf( char *fmt, . . . );
Basic conversions:
%d %ld   int decimal,long decimal
%u       unsigned int (only, no longs)
%s       string
%x %lx   int hex, long hex
%o %lo   int octal, long octal
%b %lb   int binary, long binary
%p       far pointer (in hex XXXX:XXXX)

Supported modifiers (may be combined):
%0x      Zero fill to left of number or string
%-x      Left justify number in field
%;5      Center number in field
%10x     Print number in a 10-character-wide field
%*x      Get field width from the next argument
%10.5s   Strings only: print at most 5 characters from string in a 10-character-wide
         field. If either number is replaced by a *, get the corresponding width from
         the next argument. The following are equivalent:
             printf("%10.5s", tr );
             printf("%*.*s" 10, 5, str );
```

Table 1: Printf( ) *conversion summary*

Flotsam and Jetsam

Lvalues and Rvalues

One of the most frustrating error messages to beginning C programmers is "lvalue required." An understanding of lvalues and rvalues not only gets rid of the error message, but it can also help you understand complex expression evaluations.

First some etymology. In the expression:

```
x = y;
```

x is the lvalue (it's to the left of the equal sign) and *y* is the rvalue (because it's on the right). Lvalues are single variables that you have actually declared, or to be more accurate, the lvalue is the address of a single variable that you've declared. For example, an archetypal (but stupid) compiler, when given this input:

```
int x, *p, a[10];
x = 1;
p = &x;
*p = 1;
a[1] = 2;
```

will generate the following pseudo-8086 code:

```
mov __x,1

lea t0,__x
mov __p,t0

mov bp,__p
mov [bp],1

lea bp,__a
add bp,2 ; sizeof(int)
mov [bp],2
```

Here, *x*, *p*, **p*, and *a[1]* are all lvalues because all evaluate to addresses of single objects. In assembly language, *[bp]* (but not *bp* without the indirection) and the actual labels (*__p* and *__a*) are lvalues. An array name without the brackets doesn't form an lvalue because it doesn't evaluate to a single object. In the earlier code, a *bp* (without the brackets) is an rvalue. That is, it's a temporary variable that the compiler uses on the way to doing something else.

As another example of rvalues, the expression: *x = a + (b + c);* generates code like the following:

```
mov t0,__a
mov t1,__b
mov t3,__c
add t1,t3
add t0,t1
mov __x,t0
```

The contents of the variables are put into rvalues before they're evaluated, and the actual evaluation is done on the rvalues *t0*, *t1*, and *t3*, not on the real variables. (An optimizing compiler would clean this up, of course, but it's useful to look at the way that the compiler is actually thinking.)

To understand the lvalue-required problem, consider the code generated by *i=a++*:

```
mov t0,__a
add __a,1
mov __i,t0
```

and *i=++a*:

```
add __a,1
mov t0,__a
mov __i,t0
```

(Again, an optimizer would clean things up.) In both cases the *add* instruction affects the lvalue (*__a*), but the entire expression generates an rvalue (*t0*) that contains the result of the ++ operation. It's the rvalue that's used in the evaluation of the rest of the expression (the move to *__i*).

Now consider an (illegal) statement, such as *(a+b)++*. The compiler will try to do the following:

```
mov t0,__a ; (a+b)
mov t1,__b
add t0,t1

mov t1,t0 ; ++
add t0,1
```

There are two things to notice here. First of all, the subexpression *(a+b)* generates an rvalue—a temporary variable that holds the result of the addition. Next, the last two lines aren't doing anything useful. They're just flailing around manipulating temporary variables that are never used. That's why ++ requires an lvalue, because applying it to an rvalue or anonymous temporary creates meaningless code.

As another example of how a knowledge of rvalues is useful, consider the following complicated expression, which uses *argv* as pictured in Figure 2, below (the numbers are arbitrary addresses):

```
x = * ++ * ++ argv;
```

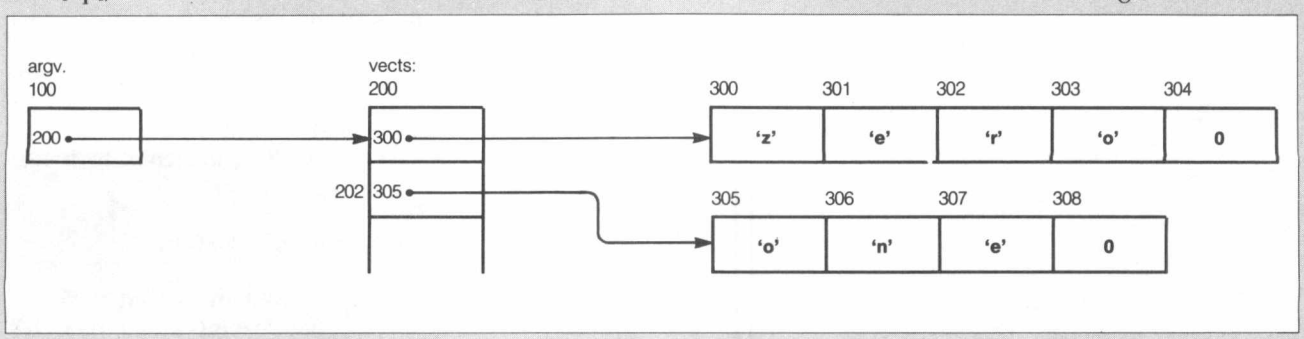


Figure 2: Argv

This expression copies the *'n'* (in "one") into *x*. To see how, think about rvalues and temporary variables. The compiler evaluates the expression from the inside out, starting at the name. The order of evaluation is determined by the order-of-precedence chart. Because * and ++ are at the same level but associate left to right, you get:

```
(x = (* (++ (* (++ argv)))))
```

The assignment is done last only because it's of lower precedence than either * or ++. The innermost *++argv* generates the following code:

```
add   __argv,2   ; sizeof(char*)
mov   t0,__argv  ; t0=200+2==202
```

so the ++ (the add instruction) was applied to a legitimate lvalue (*argv*) and generated an rvalue (*t0*) that contained *argv* after the increment. That is, given the addresses shown in the figure, the code modifies arg from 200 to 202 and then moves the 202 to *t0*. Now the compiler moves out a notch and sees the star. It applies that star to the rvalue that resulted from the previous subexpression, yielding:

```
mov   bp,t0      ; bp = 202
mov   t1,[bp]    ; t1 = 305
```

The star effectively changes the rvalue back into an lvalue (in *t1*) because *t1* holds the address of a single declared object—that is, it holds the address of *vects[1]* (202). You can tell it's an lvalue because of the brackets.

The compiler now moves out a notch and finds the second ++. Because *t1* is an lvalue, everything's OK and the following is generated:

```
add   t1,1       ; sizeof(char)
mov   t2,t1      ; t2 = 306 ;
```

Now it finds the next star and generates:

```
mov   bp,t2                ; bp = 306
mov   t3,BYTE PTR [bp]     ; t3 = 'n'
```

Finally, seeing the equal sign, it generates:

```
mov __x,t3.
```

There are two important things to notice here. First, the compiler is stupid. It always generates the same type of code for the same operator—only the names have been changed to protect the innocent. That is, a ++ preincrement always generates code having the following form:

```
add lvalue,sizeof(object)
mov temporary,lvalue
```

Similarly, a star always generates:

```
mov bp,object
mov temporary,[bp]
```

Second, the compiler always applies an operator to the temporary (or temporaries in the case of binary operators) that resulted from evaluating the previous subexpression. That is, it evaluates all expressions, no matter how complex, one operator at a time, and that operator is always applied to the temporary variable (read rvalue) that resulted from the previous evaluation. If you keep this fact firmly in mind, you can interpret any expression, no matter how tortuous it seems. —*A.H.*

C CHEST

(continued from page 100)

```
x &= (unsigned)(~0);
```

which makes no assumptions about word size.

Idoprnt( ) itself begins on line 72 (page xx). Nonconversion characters are printed by the *for* loop on line 92 and output-routine call on line 96. The *else* clause comprises the remainder of the subroutine (it starts on line 100 and extends to line 259). The % conversions are all done in this *else* clause. The various modifiers are extracted on lines 115–142, and the *switch* on lines 152–207 controls the actual conversion.

I've used a *goto* on lines 165–169 to avoid an unnecessary subroutine call. The only *goto*-less way of doing the same thing (avoiding the subroutine call) that I could think of is:

```
base = 0;
...

case 'u': base = (-10 - 16 );
case 'x': base += ( 16 - 10 );
case 'd': base += ( 10 - 8 );
case 'o': base += ( 8 - 2 );
case 'b': base += ( 2 );
```

but that's sick. (That's a technical term.) To see what's going on, note that:

```
(-10-16)+(16-10)+(10-8)+(8-2)
 +(2)                          ==-10
(16-10)+(10-8)+(8-2)+(2)       ==16
(10-8)+(8-2)+(2)               ==10
(8-2)+(2)                      ==  8
 +(2)                          ==  2
```

You can see it better if you shuffle things around:

```
-10 +(16-16)+(10-10)+(8-8)+( 2-2)
                               ==-10
```

The multiple adds are both slow and abstruse, however, so a *goto* seems a better choice.

The masks on lines 186 and 190 defeat sign extension on nondecimal *int-size* numbers and *unsigned int*s. You can't allow sign extension on a hex conversion because it would add leading Fs to the number. If the number is negative and zero fill is active, the minus sign is printed on line 200. I do it here to avoid things such as 000–2 instead of –0002. The actual conversion is done by the *ltos( )* call on line 205, which I'll discuss shortly.

The code on lines 214 to 225 both *'\0'* terminates the string and figures the length. In the case of a *%s* conversion, *bp* will hold the same pointer that was passed into the original *printf( )* call. Because the string is already terminated, all you need is the length, extracted with a *strlen( )* call on line 222. If you did the conversion yourselves, the converted characters would be in *nbuf*, and *bp* would point at the end. You can subtract the two pointers to get the length.

The remainder of the loop just prints the converted string along with any necessary leading or trailing padding. You then loop back up to get the next conversion character. The advantage of this approach is that you need only enough local buffer space to take care of the larg-

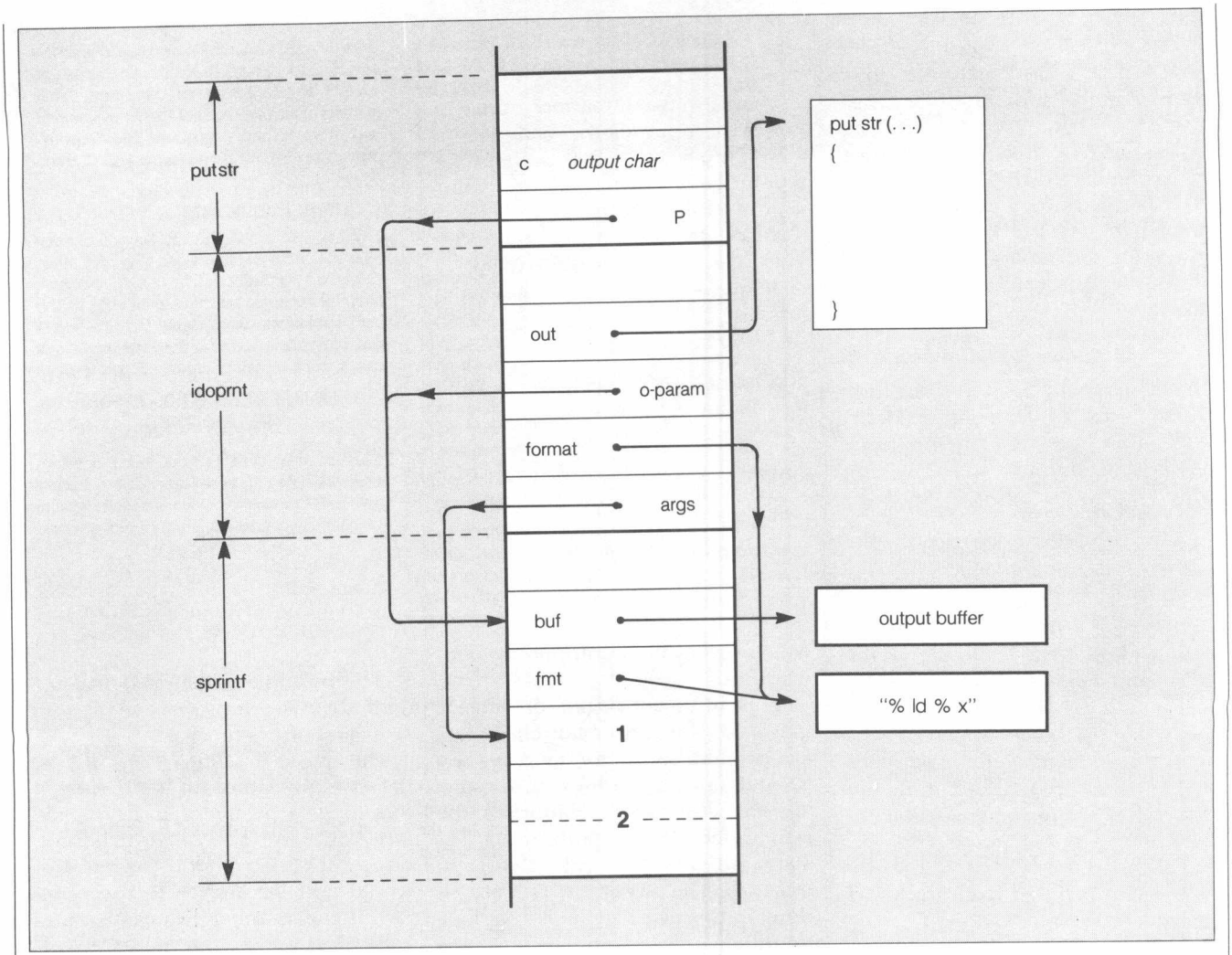


Figure 1: *Stack frame during a* sprintf( ) *call*

est possible numeric conversion, as compared to a worst-case buffer for the entire line.

The remainder of the file is just a test routine that makes sure everything works.

Ltos( ) is a *long-int*-to-string conversion routine. It's in Listing Two, page 71. There are only two things of interest. First, I'm building the string from back to front to make the conversion easier. The string is reversed in place on lines 54–61. Next is the somewhat weird-looking line 46:

```
*++bp =
"0123456789abcdef"[ n % base ];
```

The string *"0123456789abcdef"* evaluates to an rvalue of type *pointer to char* and the square-bracket notation can be applied to any pointer. For example:

```
"0123456789abcdef" [ 5 ];
```

evaluates to the character *5* (0x35).

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

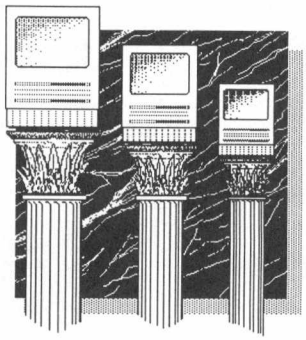
DDJ

(Listings begin on page 68.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

TO THE MACS

A Few Things That Work, Something That Doesn't, and a Little HyperTalking



by Stan Krute

In case you didn't catch my first column back in January, let me say it again: reviewing makes me uncomfortable. Creators put a lot of sweat into getting a product to market. Print is powerful stuff, and a bad review can do much harm. On the other hand, I like to share good news. So, in this column I'll talk about things I've used for a while and fundamentally like. Just call me Pollyana Krute....

On objectivity: I hate to crush any world views, but it's a myth. Product reviews are inevitably subjective. The best I can do is let you in on some of the experiential baggage I filter through. Also, I know and/or have worked with some of the people whose goodies I'll be reviewing. I'll always mention any such connections. Just know it's done to help you weigh my opinions, not as name dropping.

QUED/M 2.04

DDJ ran a nice article by Levi Thomas and Nick Turner on programmers and their text editors last year (February 1987). The article suggested that programmers might be a bit like baby ducks, bonding strongly with the first warm editor they meet. So, to help you assess the forthcoming opinions, my first computer-assisted writing was on IBM card punches and Teletype paper-printing terminals. Though primitive, both tools sported evocative sound effects.

Editing sounds have devolved, but text hacking's come a long way. I've done a lot of writing in succeeding decades, and currently work with several text-editing programs. My current favorite is QUED/M 2.04. It gives me power, speed, ease of use, and a smooth-sloped learning curve. Though I briefly mentioned the product back in column 1, I'd like to give a few more details.

Got a bad case of featuritis? QUED/M's got almost all the ones I've ever seen and/or wanted and a few more. A nonexhaustive list is:

- a powerful macro language, complete with a real-time recording mode
- high levels of user-configurability
- undos up to 32,767 (love that number) actions deep, depending on your RAM resources
- powerful regular expression and metacharacter facilities
- text folding
- line sorting
- ten clipboards
- automatic saves to two directories
- automatic saves after a configurable number of keystrokes
- backups of previous versions of a saved file
- adjustable scrolling parameters
- a full set of capitalization/casing commands
- multiline horizontal motion by tabs or spaces
- gremlin zapping
- windows with mucho status information
- window tiling and stacking commands
- user-configurable transfer menu
- parentheses balancing, with user-designatable definitions of "parentheses"
- the ability to open many files simultaneously
- multifile search/replace operations
- windows with horizontal and vertical split-paning
- text markers
- intelligent extensions to Apple's standard file-opening dialog

And on and on.... I sense QUED/M's designers have used a lot of text editors and Mac applications and have kept lists of features they've liked, detested, and wished for. And, unlike some applications that sport an abundance of commands and options, QUED/M's design and implementation seem clean, well organized, and intuitive.

There are three things I'd like in a future QUED/M:

- the ability to extend the command set via user-written standard Mac CODE resources
- the ability to work with files that don't fit whole hog into RAM
- a thicker manual, with slower/deeper explication and many more examples, particularly in the discussions of regular expressions and macros

Victor Romano programmed QUED/M 2.04. He and Jerzy Lewak designed it. Jerzy also wrote the manual. Two people, one fine text-editing product. Kudos, gentlemen, and thanks. I wish I had QUED/M in all my other computing worlds.

TMON 2.81

Here's more detail on another product I mentioned briefly back in column 1. TMON 2.81 is the Mac debugger I use all the time. It's small and rugged and works on a wide range of code resource types in a variety of complex environments.

TMON 2.81 provides a full range of standard debugging features. It runs in a simplified windowing environment that won't go down if your code munches critical parts of the Mac's regular windowing operations. In a variety of ways, you can:

- examine and change memory at

TO THE MACS
(continued from page 106)

several levels of abstraction
- stop and start code execution
- fiddle with heaps
- convert and resolve quantities and expressions relevant to Mac programming work
- customize the program by extending its command set
- recover from crashes

There is one glaring omission in this version of TMON: though it won't blow when used with a 68020 or 68881, it won't disassemble non-68000 instructions or display non-68000 registers. So you can't really debug anything involving 68020s, 68881s, 68851s, et al. Shame, shame.

TMON was originally developed as an in-house tool by ICOM Simulations, whose other products include the graphic adventures Deja Vu and Uninvited. Waldemar Horwat (not a pseudonym, he actually exists) wrote the bulk of the program, albeit at a frighteningly young age. Darin Adler, currently working for Apple's

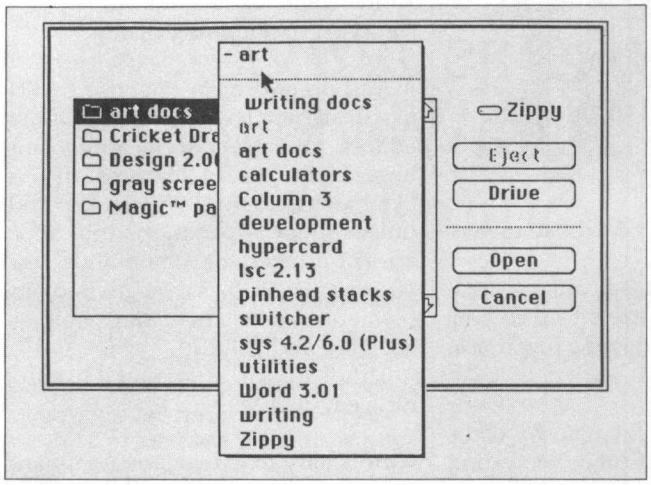


Figure 1: *A click on the directory name box in a standard file dialog brings up HFS Navigator's handy list of favorite directories.*

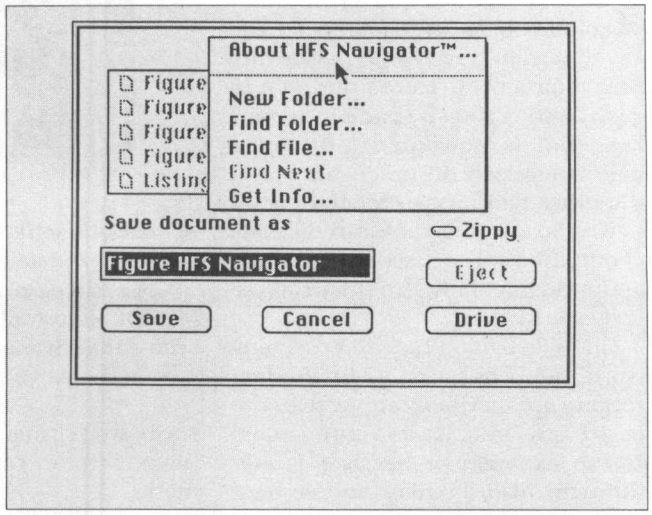


Figure 2: *Pressing the Command key causes Navigator to bring up its action menu.*

expanding Mac Tech Support team, did the set of command extensions (known as the User Area) that accompanies this version of the debugger. The new manual is quite good; Paul Snively wrote the users' guide section, and Waldemar wrote the technical reference part.

TMON may face some rough marketing pressures over the next few months as Think, Borland, and Apple bring out new source-level debuggers. TMON works primarily at the assembly level, though it has rudimentary source code label capabilities. But those other tools aren't out yet. And, even when they are, TMON should still be useful as a robust common denominator tool, especially when it's upgraded to full 680x0 support.

HFS Navigator

HFS Navigator is a small utility that Michael Kahl, chief programming force behind Lightspeed C, showed *me on my* September visit to Think central. It hooks into the Mac's standard file-opening and saving dialogs and lets you hop quickly to favorite directories on file opens and saves. It also finds files and directories and can create new directories on the fly.

That's not a bad set of features, but I already had desk accessories that pulled off the last three functions, and I've become quick and comfortable mousing up, down, and sideways through my carefully organized HFS directories. Sure, Navigator looked like a nice hack, but I didn't think I needed it.

I was (eventually) wrong. I tried Navigator for a week, gave it up, came back in a day, and now I'm hooked. You know how it is: a little taste of Mac ergonomics only makes you hunger for more, and HFS Navigator feeds the pangs.

A simple installation program lets you add Navigator to your HFS disk. Thereafter, whenever you mouse-press on the directory button in a standard HFS file dialog, HFS Navigator pops up a special menu of your favorite directories (see Figure 1, page 108). You just select the directory you want to go to, and you're there, without having to mouse up and down the branches of the HFS directory tree. The current directory, shown at the top of Navigator's pop-up menu, can either be added to or deleted from the *list of favorites*.

If you hold down the Option key when you mouse-press the standard file dialog's directory button, up comes the standard pop-up menu, showing the path to the current disk's root directory. And, if you hold down the Command key during the mouse-press, a master menu pops up that lets you find files and directories and create new folders (see Figure 2, page 108). Folder making at such an opportune moment is very useful.

I do have two things I'd like the makers to tweak, in the name of even greater ergomania. First, as currently configured, the menu of favorite directories is limited to 16 entries. I keep bumping up against that ceiling. My solution would be to raise the limit and, to keep user access fast, arrange the entries in two dimensions, as a table rather than as a list. See Figure 3, page 113,

for my vision.

Second, I don't like having to hold down the Option or Command keys when I want the standard path-to-root or master pop-up menus. A possible solution would be to give a mouse-press on one side of the directory button to get the favorites, a press on the other side to get the path-to-root, and a press within a few pixels of the top of the directory button to bring up the master menu.

HFS Navigator has a competitor, FindSwell, but I haven't got my hands on it yet. When I do I'll report back. Meanwhile, as I said earlier, Navigator's got me hooked.

WYSIWYG Ain't

I do like to maintain a positive attitude. So view the next few paragraphs as an opportunity for fabulous personal financial growth. See a need and fill it, as the kitty litter and blue toilet water inventors say.

Here's the situation: I recently spent three weeks doing production desktop publishing with a friend who's a commercial printer. I was also involved in producing the first issue of another friend's desktop-published magazine. And I just finished putting out my own little software company's latest quarterly catalog. In all three instances Macs were used, and the overall process and results were quite satisfying. But we repeatedly came up against an irritating and needless bugaboo, one that creates the incredible opportunity noted earlier.

WYSIWYG is a myth. What you see on the Macintosh screen *is not* what you get on the laser-printed page. Oh, it's close. Frustratingly close. But not close enough for real commercial production work. Hairlines vary in size. Text and graphic elements lose their relationships. Whole sentences can switch pages. On-screen measurements indicate you're controlling placement to four or five decimal places, and the finished result can vary by big pieces of inches. And this is true, to varying but never completely insignificant degrees, with *every* piece of desktop software we used. So you proof and kludge and proof and kludge until the result's acceptable.

There's no good reason for such behavior. These are computers, kids, and they're very good at arithmetic. The Mac has SANE, with ungodly levels of precision. If software SANE's too slow, you can write directly to the hardware. If Apple's LaserWriter driver doesn't work properly, just send PostScript out directly. If PostScript's the problem, learn how to work around it. But, please, somebody, do something. This is intolerable.

Deliver desktop publishing accuracy along with ease of use and automated power, and you'll get very rich. 'Cos there are a lot of people out there doing this stuff, and they all know the truth: the WYSIWYG emperor's walking around stark jaybird nekkid!

Code Corner

I spent some time this last month wandering around HyperCard and HyperTalk. Serendipitous travel's my most fruitful learning mode. This month's code corner features a small toolkit I built to aid that exploration. Building and running it revealed some of HyperCard 1.0.1's contradictory qualities.

First, the script editing facilities

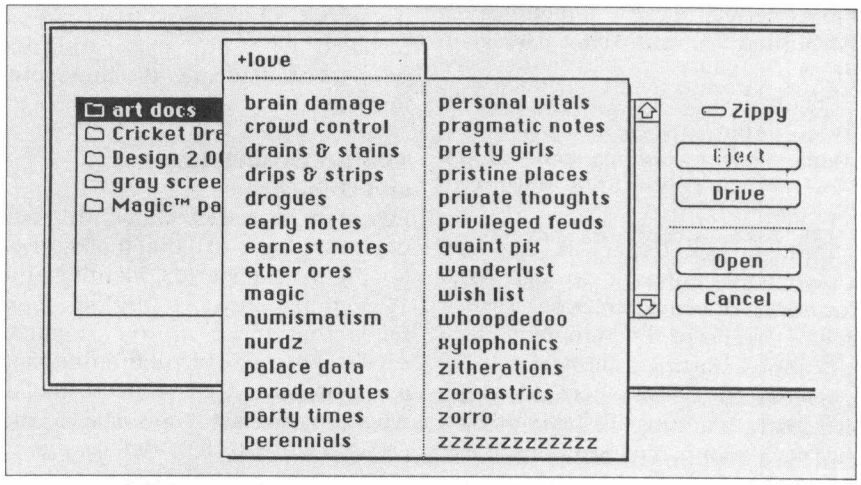


Figure 3: *Author's vision of an improved Navigator, stretching the user interface along an orthogonal path*

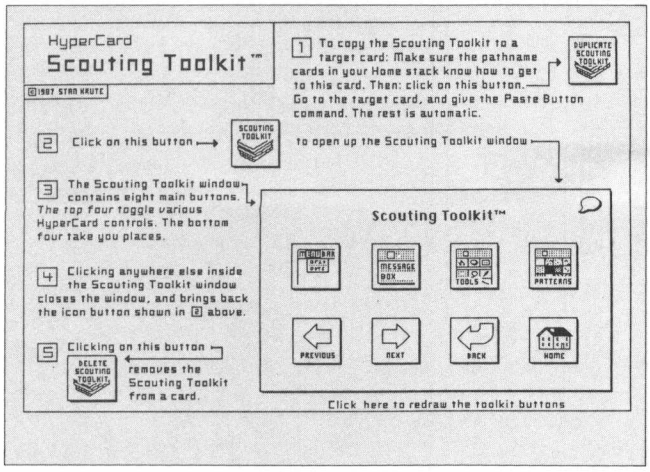


Figure 4: *The Scouting Toolkit's card of residence*

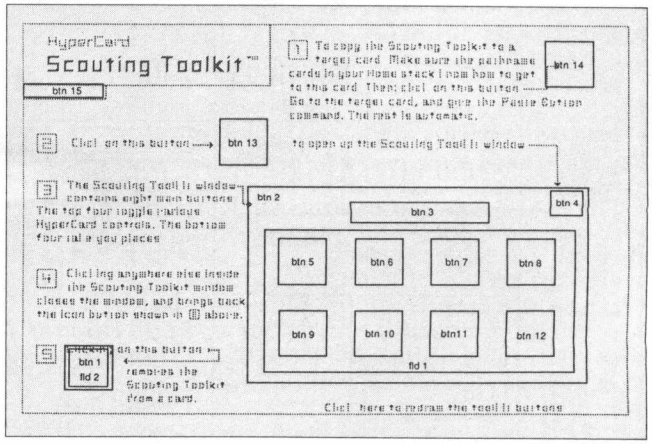


Figure 5: *Where the 15 buttons and 2 (hidden) fields live on the toolkit's residence card*

can be politely described as primitive, but the graphics editing facilities are elegant.

Second, execution speed can be slow. Depending on the hardware, this month's project takes 0.5 to 4 minutes to copy less than 20 objects from one stack to another. Third, execution speed can be fast. My buddy Bruce The Q. Hammond built a 40,000 card HyperCard database (he likes to break and fix things). Searches over this multimegabyte file were in the sub-30-second range.

Fourth, HyperTalk syntax can be intuitive. Right from the start I could write large pieces of code without continual manual browsing. Fifth, HyperTalk syntax can be inconsistent—for example, the usage of the word the. Sometimes it's optional, sometimes it's mandatory, sometimes it's forbidden. Aargh! Who can keep track?

Sixth, HyperCard can be used to build large, complex coordinating tools. Take a look at some of the commercial applications starting to appear.

And finally, HyperTalk can bog down while running through large amounts of complex code. Take a look at some of the commercial applications starting to appear.

Notice the seesaw here. It's good, it's bad. It's fun, it's irritating. It breaks Macintosh conventions, it extends the Macintosh metaphor. Hmm. . .final judgments will obviously have to wait. Let's just hope Bill and his team keep at it 'til they get it right. Meanwhile, on to the project.

Getting a Toehold

As I learned long ago from the Kernighan/Ritchie/Plauger crowd, the first thing you want to do in a new environment is build some simple tools, then lever yourself up into power and sophistication. HyperCard's no exception to the rule.

Whenever I started work on a new stack or card, I found myself spending quite a bit of time bringing in the basic buttons that let me hook into HyperCard's facilities. I'd find myself unable to go Home or without a menu bar. And I hate memorizing key combinations. So, I built a useful set of buttons that can be installed with one click and one paste—automatically. It's the Scouting Toolkit.

Toolkit Descriptions and Behaviors

Figure 4, page 113, shows the self-documenting card the toolkit lives on. Figure 5, page 113, identifies the 15 buttons and two (hidden) text fields that make up the domicile card's object world. All but one field and one button go with the toolkit when it travels to new stacks and cards. Figure 6, left, details the 12

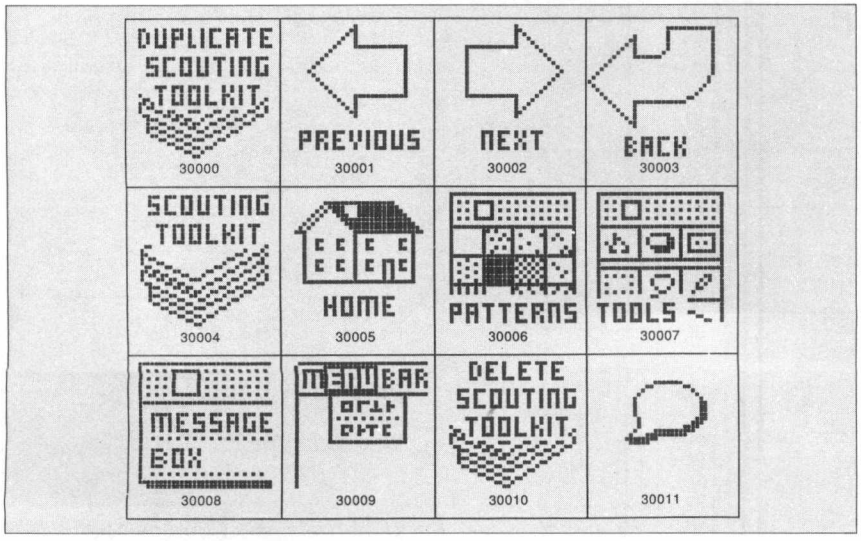


Figure 6: *Design details of the 12 icons used in the Scouting toolkit*

icons used in the buttons. Listing One, page 72, gives complete descriptions and scripts for the card, its stack, the buttons, and the fields.

Installing the toolkit on a new card is easy. First, make sure your Home card knows where the toolkit's stack lives. Then click on button 14. It copies itself to the clipboard. Now go to your target stack and card, and give a Paste command. As ceaselessly mentioned, the rest is automatic.

The toolkit proper contains eight major buttons, numbers 5 through 12. The top four let you toggle various environmental windows. The bottom four take you to useful places. The other toolkit buttons support the big eight.

The whole toolkit folds up and disappears if you click in its interior outside the buttons. It reinflates when you click on its iconic remainder, button 13.

Like all good organisms, the toolkit knows when to fold its tent and scoot off into the night. Button number 1 removes all traces of the toolkit from a card, including itself. Snake tail swallowing is one useful byproduct of HyperTalk's ability to self-reference.

Trees Saved As Description's Delayed

I keep telling Tyler I'll cut down the size of these columns and their listings. And I'm trying. But once again I've exceeded my spatial budget. So you'll have to wait until next month for the remaining discussion of the Scouting Toolkit's objects and their operation.

Of course, Marvel and DC always come up with dramatic catchphrase hooks for their continued graphic stories. But all I can think of is this small section's subhead. Who ever said comic books ain't superior literature?

Wrap-Up

I've worked without reader feedback on these first few columns—frightening but true. So, if you get a chance and have the interest, drop me a note detailing what you want to see more and less of.

Next time out, besides the project wrap-up: reader mail, Mac Expo, parentheses surrender, the Cambridge ambience, intelligent pictures, Microsoft madness. And, of course, another code project. See you in 31.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 72.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

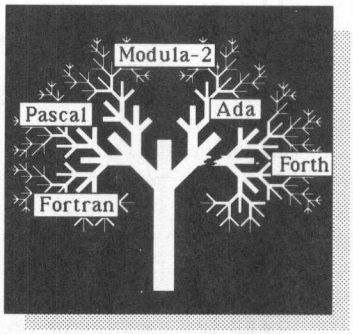
Vendors

HFS Navigator
Think Technologies
135 South Rd.
Bedford, MA 01730
(800) 648-4465
(617) 275-4800
Reader Service No. 20

QUED/M 2.04
Paragon Concepts Inc.
4954 Sun Valley Rd.
Del Mar, CA 92014
(619) 481-1477
Reader Service No. 21

TMON 2.81
ICOM Simulations Inc.
646 S. Wheeling Rd.
Wheeling, IL 60090
(312) 520-4440
Reader Service No. 22

Implementing Wirth's LineDrawing Module



In his seminal work *Programming in Modula-2*, the venerable Nicklaus Wirth, lord of structured programming, proposes a graphics module that he calls LineDrawing. It's defined on pages 114–115 of my copy, which is the Springer-Verlag 1982 edition. Although LineDrawing lacks the razzle-dazzle of more recent graphics packages, it's serviceable for many applications. Yet Modula-2 vendors seem to have bypassed it even though they leap at everything else Wirth even vaguely suggests. Consequently, this month's column makes a Valuable Contribution by implementing LineDrawing and showing some ways to use it.

For this project I tried out a new Modula-2 development system from Stony Brook Software up in Wilton, New Hampshire. I used Version 1.00, a level number certain to make any serious programmers break out in a sweat as cold as the New Hampshire winter. It's a command-line compiler, which isn't much fun after getting used to Turbo Pascal 4.0's glitzy integrated environment. Also, I didn't use the included program editor, M2EDIT, because I'm spoiled by BRIEF. (If you haven't tried BRIEF, it's like coming to the One True Religion.) Still and all, Stony Brook's Modula-2 is a terrific compiler. I found only one real bug using it for this and some other projects: remarkable for any new product. It's dazzlingly fast as well, compiling the 201-line LINEDWG.MOD in five seconds from Enter to system prompt on my 8-MHz AT clone with a 40-ms hard disk.

by Kent Porter

Lest the previous paragraph leave you with the impression that I'm just overflowing with praise for all and sundry, let me take a whack at The Master Himself. Professor Wirth might be the progenitor of structured programming languages, but he writes stylistically sloppy code. If this guy were a student in one of my programming classes, I'd drop his grade for the lack of comments and a tendency to put unrelated statements on the same line. Wirth gets away with it because he's Who He Is, but that's no excuse, particularly in a work that is to Modula-2 what K & R is to C. And while I'm at it, Springer-Verlag gets a hiss for consistently producing the worst-indexed books in the computer publishing industry.

So much for congratulations and contumely. Let's get on with it.

The Line-Drawing Module

The definition module for LineDrawing is LINEDWG.DEF in Listing One, page 88. With minor stylistic changes, it adheres faithfully to Wirth's definition. The only alteration of substance is the name itself; LineDrawing doesn't fit into the eight characters DOS allows, so I shortened it to LineDwg.

There's one other change as well. In the prototype, Wirth defines a procedure called *area*, which fills a rectangular region with a color. But elsewhere, in connection with the Queens program on pages 58–59 (which actually calls the procedure), he refers to an apparently identical process called *paint*. The latter makes more sense, so that's its name in LINEDWG.DEF.

I chose to implement LineDwg using EGA (the IBM Enhanced Graphics Adapter) mode *10h*, which furnishes 640 × 350-pixel color graphics. This is somewhat at odds with the Wirth model, which proposes four gray scales from white (0) to black (3), but it makes sense in that the EGA is widely available and it has decent resolution. *Mapping* Wirth's color indicators to EGA color numbers is easy to do via a *CASE* statement.

The problem with EGA 640 × 350 is that the pixels aren't square. To achieve orthogonal integrity on an

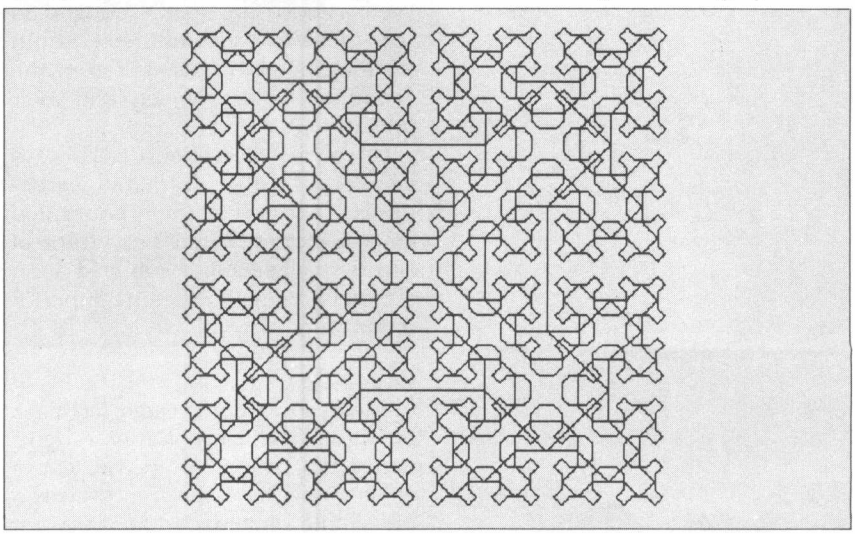


Figure 1: *Output from SIERPIN.MOD, drawn with LineDwg routines*

IBM PC display, the *y* dimension should be 75 percent of the *x*. The EGA resolution yields 53.8 percent of *x*, so everything is taller than it should be. Because Wirth's line-drawing procedure works in increments of 45 degrees, he obviously had square pixels in mind.

The solution is to use a virtual-coordinate space to represent the display. In virtual space, the screen can be 800 × 600 pixels, and following the Cartesian convention, the origin (coordinates [0,0]) can be at the lower-left corner with y ascending upward. Because the virtual range is larger than the device range, rounding off tends to jam pixels together, thus producing visual objects without gaps. The opposite effect—gapping—occurs when the virtual range is smaller than the physical, as in 400 × 300 pixels mapped to a 640 × 350-pixel display. In both cases you get jaggies, but jaggies are an inevitable feature of the computer graphics landscape. Fortunately, they're not glaringly apparent with the EGA.

Mapping virtual coordinates into physical space is relatively simple. The program works entirely within the virtual range, and only the output functions "know" that the coordinate system is a myth. The functional procedures *devX* and *devY* in Listing Two (the LineDwg implementation), page 88, perform virtual-to-physical translations based on factors computed during the module's initialization phase. These functions are called when writing pixels to the device. The only procedure exempt from virtual addressing is *writePixel*, which works entirely in device space.

Perhaps one reason why LineDrawing has not been implemented elsewhere is that Wirth's thoughts on the subject seem to be incomplete. For example, he formulates the *PaintMode* enumeration as a control for the *paint* and *copy* routines, yet none of the calls ever touches it. Consequently, it's set here to *replace* and never changed, and the routines don't refer to it. Likewise the matter of color inheritance is unresolved. The *paint* and *dot* procedures accept a *color* parameter, but what color applies to *line*? I opted for a "last color used prevails" strategy. The *color* variable, global to the implementation module but not externally visible, forces the next line to inherit the color of the most recently written pixel.

Wirth's approach emulates turtle graphics by employing the concept of a graphics pen that draws a line as it moves, and any activity causes the pen to come to rest at the last pixel written. You can see this idea at work by examining the *line* procedure. *Px* and *Py* are externally visible virtual coordinates giving the pen's position. *Line* advances them before writing each pixel so that, when the line is completed, they reflect the location of the most recent pixel. A pen-up motion is effected by changing the values of *Px* and *Py*; the turtle instantly moves without drawing. The same end results from writing a black pixel (color 3) to a nonsequential location via a call to *dot*.

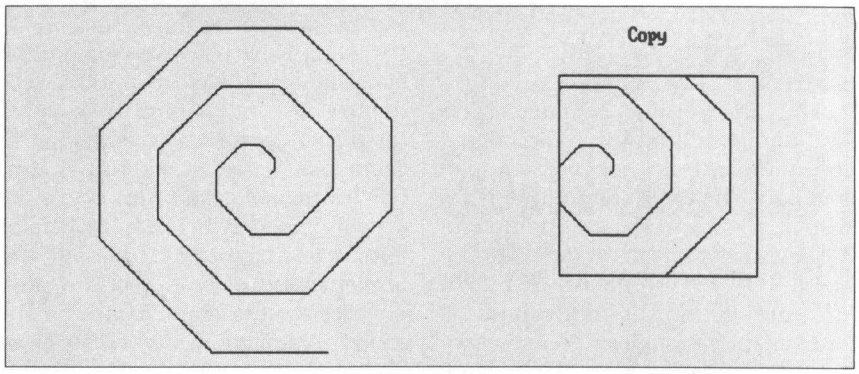


Figure 2: *Output from SPIRAL.MOD (Listing Four)*

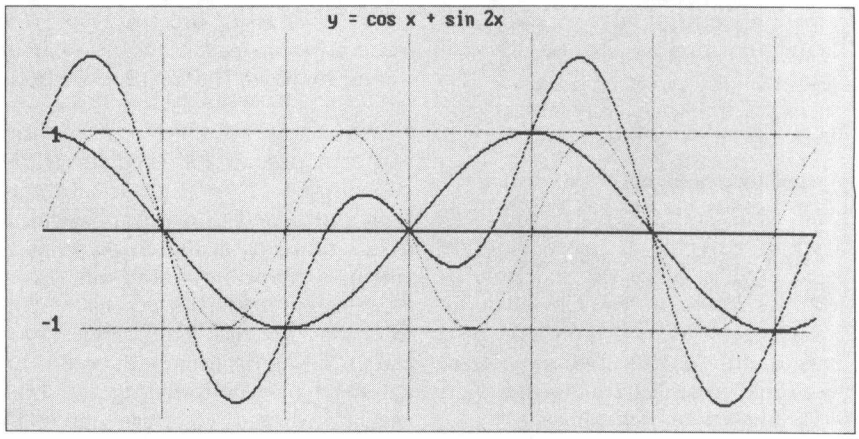


Figure 3: *Photo of MathPlot's output*

STRUCTURED PROGRAMMING

The *line* routine in LineDwg is quite limited. It draws in direction *d* for *n* units from the current pen location. The direction indicator *d* is a digit 0..7 expressing a multiple of 45 degrees such that 0 draws to the right, 1 to the northeast, 2 straight up, and so on. To draw lines at unpredictable angles, it's necessary to develop your own routine outside LineDwg; it should call the *dot* procedure for output because *dot* maps virtual coordinates into the physical space.

The *clear* procedure does more than simply clear the screen, but it's consistent with Wirth's ideas. On the IBM PC, any mode change clears the display, even if you switch to the same mode that's currently active. Wirth's demonstration programs inevitably begin with *clear* in lieu of initiating a graphics mode, so the implemented *clear* procedure puts the display into graphics. In the absence of indications otherwise, *clear* also establishes white as the default color. Wirth never calls *clear* again after the start of the run, but your programs can call it any number of times to reset the display without leaving graphics mode.

Using the Module

From the standpoint of the PC, Wirth's programs are inconsiderate because they leave the machine in graphics mode on exit. Obviously he doesn't program on a PC, or he'd have included the switch back to text that precedes the termination of Listing Three, page 90.

This program, called SIERPIN.MOD, is included here as a test to make sure the graphics functions work as expected. Wirth's book shows the output of the program, which is based on a mutually recursive graphics algorithm developed by the Polish mathematician Sierpinski. The only changes from the Wirth model are the addition of the switch back to 80×25-pixel color text at the end and removal of superfluous *Read* statement from the *REPEAT...UNTIL* loop. The output of the program (to my astonishment, it ran correctly the first time) appears in Figure 1, page 118.

The *paint* procedure fills a rectangular area with the specified gray scale. The origin of the area is at (*x*, *y*), extending right for *w* units and up for *h* units, expressed in virtual coordinates. This routine is internally optimized to prevent rewriting pixels that physically coincide as a result of mapping round-offs. Even so, because it's based on ROM BIOS interrupt *10h*, it's agonizingly slow. Much greater efficiency could be achieved by manipulating the EGA registers directly. I chose not to in the interest of limiting the scope of this project; the EGA is a whole big subject of its own.

The *copyArea* procedure also uses the ROM BIOS and is similarly poky in performance, even though—like paint—it's internally optimized to operate at the physical pixel level. This routine copies part of the screen to another location. The source is at (*sx*, *sy*) and the destination at (*dx*, *dy*), the copied area being *dw* virtual units wide by *dh* units high. You'll see it in operation a little later.

The two text routines in *LineDwg* are *Write* and *WriteString*, which parallel similarly named procedures in Modula-2's standard InOut module. The difference is that they start writing at the current pen position—well, near it, anyway. This implementation uses the default EGA character set rather than a stroked font, so it places the text cursor—hence the first character—in the 8×14 physical cell occupied by the pen. That's as close as you can get with the built-in characters, and it should be good enough for most applications. The foundation routine is *Write*, which outputs a single character and advances the pen. *WriteString* merely calls *Write* for each character in the string.

This is where the bug in Stony Brook Modula-2 popped up. It's unclear what's wrong, but literals passed as parameters to *WriteString* sometimes show up with a garbage character appended to the end. Either the compiler isn't placing a null terminator in literals, or else the *Length* procedure has a glitch. Through trial and error, I discovered that an extra space at the end of the literal makes the problem go away sometimes, and other times taking that padding space out has the same effect. This is a bug in the category of an annoyance and hardly a major flaw.

The SPIRAL.MOD program in Listing Four, page 92, exercises all directions of *line*, plus *copyArea* and the text-writing routines. Though a small program, it runs for quite a while because of the inefficiency of ROM BIOS calls, two of which are made for each pixel copied to the outlined window at the right side of the screen. The figure copied is a portion of the spiral pattern; Figure 2, page 122, shows the final screen.

These two demonstration programs are nice to look at, especially the interesting Sierpinski curve, but they have no practical value. For that reason, I've thrown in MATHPLOT.MOD (Listing Five, page 93) to illustrate a more useful application of Wirth's LineDrawing module. This program plots the function y = cos x + sin 2x, which is representative of any number of mathematical graphing applications in both business and scientific programming.

MathPlot draws three curves in different gray scales using the *dot*

routine. Dark gray follows sin 2x, light gray plots cos x, and white is the curve deriving from their sum, where $0 < x < 10$. Thus the curves go slightly beyond $x = 3\pi$, tracing the function through a bit more than 1.5 circles. The vertical grid lines delineate intervals of 2π, and the horizontal grids indicate unity on either side of the x axis, giving visual reference points for the curves. The photo in Figure 3, page 122, doesn't do it justice. This is a handsome display that you have to see on an EGA screen to appreciate the usefulness and elegance of the gray-scale plots produced by Wirth's LineDrawing module.

Stony Brook's Modula-2

This experience with Stony Brook's Modula-2 sold me on the product. Guys like me, writing for several computer magazines and on a first-name basis with the Fed Ex man because he brings so many review copies, tend to get jaded. It's hard to work up enthusiasm over yet another new product in an industry that sprouts them like weeds on a vacant lot. So praise doesn't come easily.

I'm not saying it's perfect. In addition to the bug I mentioned earlier, I found a truly ugly glitch while developing MathPlot. In the *yc* procedure, I originally wrote *INTEGER (y * 100)*. This is a dumb mistake in that a type transfer (or cast in C parlance) isn't a true function and thus doesn't resolve parametric expressions. I just forgot, and I'm not ashamed; programmers make such mistakes all the time. But it gave the compiler a nervous breakdown, causing it to generate a screenful of cryptic internal error messages. I spent half an hour commenting out program lines one by one before discovering the offender. And in fixing that problem, I found an error on page 114 of the manual, which says that the *entier* function (*REAL* to *INTEGER*) returns a *REAL*, which is clearly wrong. There are probably a few others of this sort that I haven't uncovered yet.

But overall, Stony Brook Modula-2 is good. It doesn't have quite as many bells and whistles as its better-known competitor from Logitech, but the tools it provides are well rounded: 23 built-in modules, an editor, a MAKE utility, and a symbolic debugger. It supports the usual array of memory models, has a satisfying selection of compiler options, and uses the standard DOS linker. Above all, it's fast. If you program in Modula-2, a viable and less intimidating alternative to C as a systems programming language, you'll like it. I sure do.

Do you have a programming problem you'd like to see addressed here? If so, drop me a line at *DDJ* (no calls, please!) or leave a message for KPORTER, Mountain View, CA, on MCI Mail. No promises, but all suggestions are welcome; dreaming up a subject for a monthly column is a fast track to burn-out.

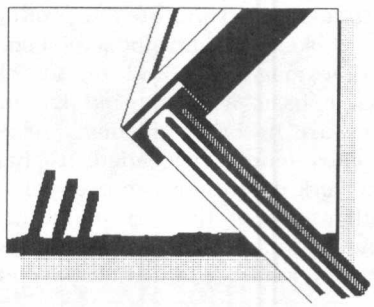
DDJ

(Listings begin on page 88.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

THE FORTH COLUMN

by Martin Tracy



The Ninth Forth Modification Laboratory (FORML) Conference was well attended and well worth going to. It was held on a beautiful Thanksgiving weekend in Monterey, Calif., and you could watch the pelicans migrating southward along the Pacific Coast and see swarms of Monarch butterflies hanging from the trees.

As usual, there were two days of solidly packed presentations, with wine and cheese in the evenings. Stephen Sjolander presented a Novix 4016 decompiler that would be of great value to anyone working with this chip. Bob La Quey pointed out that Forth would be an excellent language for implementing Hypertext. George Shaw presented the most complete and comprehensible paper on QUANs (multiple code field words) that I have ever seen. Stephen Pelc showed an implementation of record and field defining words, and Dr. Ting shared a short recursive line-drawing algorithm. This year, Dr. Ting presented even more papers (four) than Wil Baden (three)!

There were many other high-quality papers that you will be able to read in the *1987 FORML Conference Proceedings*, available from the Forth Interest Group ([408] 277-0668) later this year. Speaking of that, the *Proceedings of the 1987 Rochester Forth Conference* is now available as *The Journal of Forth Application and Research* (*JFAR*), vol. 5, no. 1, 1987, from the Institute of Applied Forth Research, 70 Elmwood Ave., Rochester, NY 14611. Or you can order it indirectly from FIG.

The informal theme of FORML this year seemed to be *BLOCK*s vs. text files. Each attendee received a copy of Tom Zimmer's PF Forth (formerly ZF Forth and soon to be FF Forth). Tom's IBM PC public-domain dialect is the joint effort of Zimmer, Smith, Curley, and Modrow. It is enormous by Forth standards, occupying 1 Mbyte of disk space. It includes source files for dozens of utilities. You can get an ARCed copy from the GENie Forth bulletin board (see *DDJ*, December 1987). Expect to spend a lot of weekends learning it.

PF Forth keeps all its source in text files and includes a sophisticated file editor. The program image is in a .EXE file, and headers are kept in a separate segment. It is closely related to Mike Perry's unpublished F83Y. Mike is the Perry behind the Laxen/Perry F83 dialect. (Henry Laxen has, for the moment, died and gone to C.)

In Mike's own words: "In an additional attempt to merge with the environment, explore the trade-offs, and generally ingratiate myself with certain fanatic elements, I discarded BLOCK. Don't worry, it's still available as a loadable option. When a file is loaded, it is read into the current file segment with a single call to DOS, then interpreted. Note the 64K limit. All control characters and blanks are treated as white-space. This allows a single compare to be used. Compilation is quite fast compared to whatever I used to use."

The first ever Australian Forth Symposium will be held May 19-20, 1988, at the NSW Institute of Technology. The focus is on productivity, and there will be papers, demonstrations, and hands-on instruction. Chuck Moore will be there. If you want to go, ring Jose Alfonso or Dr. Walker on (02)20930 or Roy Hill on (02)217-3828.

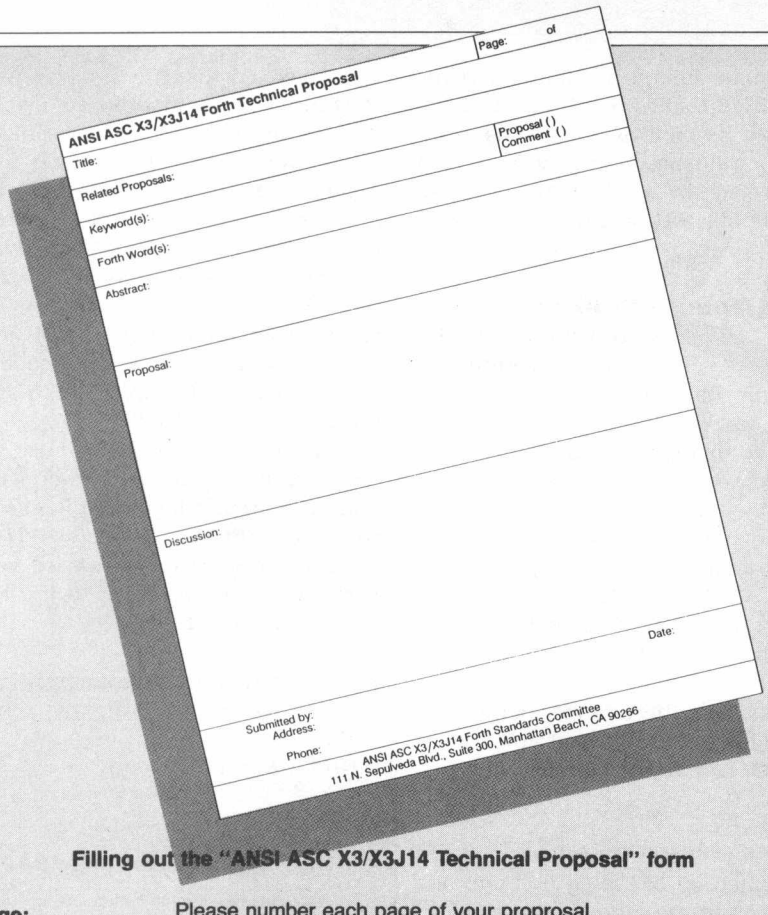
The next ANSI X3J14 Forth standards meeting is also in May, somewhere in the Northeastern United States. Contact me for details if you would like to attend. You can reach me at Forth Inc., 111 N. Sepulveda Blvd., Manhattan Beach, CA 90066.

At the second ANSI Forth meeting (November 1987), preparations were made for accepting technical proposals. A copy of the approved proposal form is on page 132. About two dozen proposals have been received and many more are sure to follow. There seem to be roughly a dozen technical issues: vocabularies, mass storage, flow of control, arithmetic, documentation requirements, testing, assemblers, controlled reference word set, ROMability, host and file structure, the interpreter, Tick >BODY and addressability, numeric output, and exception handling.

Each member of the ANSI Forth committee will prepare a position paragraph on each issue. One member, the "magnet," will collect all the paragraphs and synthesize them into one paper describing the different sides of the issue. You can follow or participate in the discussion on any issue by signing onto the GENie Forth bulletin board, Category 10, Forth Standards. I'm told that several hundred people have already perused this category.

Two New Books on Forth

You will find the new *Dr. Dobb's Toolbook of Forth, Volume II*, to be an excellent anthology of papers on Forth programming techniques. Inside, you can find Bresenham's algorithm, FFTs, spreadsheets, and Logo, all implemented in Forth. Some papers are taken from FORML and Rochester proceedings, and others first appeared here in *DDJ*. Unfortunately, credit is not given to the



Filling out the "ANSI ASC X3/X3J14 Technical Proposal" form

Page:	Please number each page of your proprosal.
Title:	A short phrase that characterizes your proposal. Example: String extensions.
Related Proposals:	If you have submitted other proposals on this subject, please list their titles and dates.
Proposal( )	Are you proposing a specific change? Then check the Proposal box.
Comment( )	Otherwise, check the comment box.
Keyword(s):	Pick a keyword that helps others understand what area of Forth your proposal addresses. For example: Kernal, double integers, system word set. . .
Forth word(s):	List all affected Forth words, including ones added or deleted by your proposal or discussed in your comments.
Abstract:	Briefly convey the nature of your proposal (or comment).
Proposal:	State your proposal in specific terms. When proposing a new word, or changes to an existing word, simply state the new definition. Include a stack picture if appropriate.
Discussion:	If you are making a comment, put it here. If you are submitting a proposal, provide compelling arguments in favor of your proposal. Be concise. If you are aware of arguments against your proposal, state them and rebut them. Is your proposal consistent with current Forth conventions? Does the proposal involve hardware or other system dependancies? Does if affect application portability? Does it have implications for multi-user environments? Should the proposed change by mandatory? Does it affect ROMability? Is it general purpose? How does it affect execution speed? How does it affect compilation speed? What are its memory requirements?
Submitted by:	Provide your name, address, and daytime phone number.

Use the "ANSI ASC X3/X3J14 Technical Proposal Form, cont'd" form if additional pages are needed. Remember to put page numbers and repeat the "Submitted by" and "Date" fields on each page.

Figure 1: *Approved ANSI Forth proposal form*

(continued from page 130)

original publication. There are 32 quality papers in all, many of them with source code. The Toolbook sells for $29.95 and is available from *DDJ* ([800] 533-4372 or [800] 356-2002 in California). If you don't want to type in the programs, you can get them on a disk for an additional $25.

Dick Pountain's *Object-Oriented Forth* (Academic Press, 1987; $19.95) begins: "Forth is, regrettably, one of the best kept secrets in the computing world." In this book, Dick extends Forth to include named objects, which increases its readability and reusability. In essence, an object is a defining word used to create defined words called instances. An instance has its own hidden data structure, made of instance variables, and a set of permitted operations, called methods. Instance variables are to instances as user variables are to tasks.

When an object is defined, all instance variable and method names are removed from the Forth dictionary by relinking the headers. Only the object itself can find them because it has the initial link, or "key," to the hidden vocabulary. All instances of this object are state smart and know the key. During compilation an instance looks the following method up in its private dictionary and compiles a reference to it.

As an example, consider complex numbers as objects. They might have instance variables for the real and imaginary parts and methods for addition, multiplication, and so on. Pountain limits his methods to @ and !, however, so that complex results are left where they belong, on the stack:

```
TYPE> COMPLEX
   2 VAR REAL
   2 VAR IMAG
OPS>
   : COM! IMAG ! REAL ! ;
   : COM@ REAL @ IMAG @ ;
ENDTYPE> COMPLEX

COMPLEX X 2 3 X COM!
```

Object-Oriented Forth (*OOF?*) takes you through this and other abstract data types, including lists

and heaps. This book is not for beginners, and I highly recommend it.

Dead Duck's Hard Beak

Starting in the October 26, 1987, issue of *InfoWorld*, Steve Gibson ran three articles in his Tech Talk column describing the implementation of Microsoft's QuickBASIC 4.0. According to Mr. Gibson, "The technology is known as 'Threaded PseudoCode' ... and was first seen in the strange powerful language Forth." He continues, "Forth is the language you want to pretend doesn't exist because it's horrible to write with but hauntingly powerful."

Needless to say, these comments drew immediate rebuttal from the Forth community. Here is a sample, translated from Timothy Huang's Chinese Forth column: "... the term Threaded PseudoCode was used repeatedly with such an envious attitude, like a human being who just discovered the warmth of the sun. In fact, we have been using this technique for more than ten years. So why doesn't Mr. Gibson and Microsoft, as well as the whole computer industry, come straight out and recognize FORTH, and quit behaving like a dead duck with a very tough beak?"

Text from BLOCKs

In my last column, February 1988, I presented command for reading and writing data and text files from *BLOCK*s. Some of the source screens needed fixing, and you will find corrected versions in this month's Listing One, page 94.

The new *GETTEXT* includes the flow-of-control construct *BEGIN... WHILE...WHILE...REPEAT*. This construct occurs naturally in both string and file applications:

```
BEGIN ( Are there more characters?)
WHILE ( Is this one that you want?)
WHILE ( Go ahead and use it.)
REPEAT
```

Note that this is a proper structure, delimited by *BEGIN* at one end and *REPEAT* at the other.

In most Forths, the stack is used at compile time to hold fix-up addresses until they can be resolved. In some dialects, such as polyFORTH and ZEN, only these addresses are on the stack. Furthermore, *WHILE* swaps addresses to keep the *BEGIN* address on top. *BEGIN...WHILE...WHILE...REPEAT* can be implemented simply by adding *THEN* to resolve the second *WHILE*, as in *BEGIN... WHILE... WHILE...REPEAT THEN*.

Other Forths, especially those descending from FIG Forth, keep extra flags on the stack for "compiler security." If *BEGIN...WHILE...WHILE ... REPEAT* does not work for your Forth, you can change it to the less elegant construct:

```
BEGIN ( Any more characters?)
DUP IF DROP ( Want this one?)
THEN
WHILE ( Eureka! Use it.)
REPEAT
```

DDJ

(Listing begins on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

EXAMINING ROOM

Turbo Professional 4.0

Product:
Turbo Professional 4.0
Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later; Turbo Pascal 4.0 or later; hard disk recommended
Pricing:
$99
Vendor:
Turbo Power Software, 3109 Scotts Valley Dr., Ste. 122
Scotts Valley, CA 95066; (408) 438-8608

If you want to protect your reputation as a guru, don't let anybody know you use the Turbo Professional 4.0 library. The guys down at Turbo Power Software have made the hard stuff so easy that it's bound to reduce your fellow programmers' awe of your coding prowess.

Imagine! You can create a pop-up window with a one-liner, and with other one-liners you can rearrange a stack of windows to bring the selected one to the top, make any of them go away, and write to the active window. Or you can similarly manipulate EMS and pull-down menus, insert chained interrupt handlers, and manage TSRs.

All kidding aside, Turbo Professional 4.0 is a superbly crafted toolbox for serious programmers using Turbo Pascal 4.0. Consisting of some 300 functions and procedures spread across 25 Pascal units, it truly does render the difficult accessible. Some of these routines address defects in Turbo Pascal 4.0—lack of a BCD type, for example—while others simplify the dirty work of advanced applications development. My particular favorite, the unit on TSR management, removes the mysteries and perils from this most frustrating kind of program. There are even functions for changing the hot key on the fly and disabling a TSR. The manual has six lucid pages of discussion that make TSRs make sense. The 426-page manual does a workmanlike job of explaining how all these abstractions of complexity fit together and operate, and what the parameters mean.

There isn't room here to cover all the goodies in this package, so I'll list the major categories and discuss a few particulars. These are the groupings of routines:

- Screen: Virtual screens, popups, pulldowns, input editing
- Strings: Manipulations, formatting, long ASCIIZ, DOS command line parser
- Low-level: Sugar-coated DOS and ROM BIOS calls
- Interrupts and TSRs: ISR and TSR management, 8087 TSR support, critical error handler
- Sorting: In-memory non-recursive quicksort of up to 64K elements
- Memory: EMS and extended memory management
- Macros: Keyboard macros and an editor
- Large arrays: In RAM, EMS, and virtual memory
- BCD: 18-digit precision for arithmetic and transcendentals
- Other: Runtime error recovery, inline macros

Additionally, the package includes about a dozen demo programs that really are useful, such as a programmer's calculator (a TSR written with the library), time management/billing utilities, and a GREP-like text finder.

Turbo Professional comes on three diskettes, only one of which (the TPU disk) you need to copy to your Turbo Pascal 4.0 directory in order to use the software. The other two disks contain the demo programs and archived source code for the entire library, along with an unarchiving utility.

You harness the horsepower of this programming engine with the USES statement. All the units have the name pattern TP*.TPU, which makes them easy to identify. There are some unit dependencies, and several depend on Pascal's DOS unit as well. Appendix A in the manual lists them. As an example, to do virtual screens, the directive is

```
USES DOS, TPCRT, TPScreen;
```

The syntax of some of the calls is a little awkward, but that doesn't diminish their power. For example, to initialize and work with a pop-up window, the call is

```
IF NOT MakeWindow (win1, . . .)
THEN
  {Error occurred}
ELSE
  {Do window stuff}
```

(MakeWindow is a function taking 12 arguments.) If MakeWindow is successful, you can pop it up with

```
BoolResult := SetTopWindow
(win1);
```

where BoolResult becomes TRUE if the operation works. Getting rid of the most recently popped window is a different matter:

```
winPtr := EraseTopWindow;
```

Ron Copeland, associate editor, is the coordinator for this review section. He welcomes your feedback on products worth reviewing.

(continued from page 136)

which returns a pointer to the window erased. The extensive use of such value-returning functions rather than more intuitive procedures gives the library a C-like feel.

The BCD module brings 18-digit precision to Pascal. This is much better than the 11-maybe precision of the REAL type and not subject to roundoff errors or the unexpected appearance of E notation. BCD is usually associated with business programming, but Turbo Professional makes it viable for scientific applications as well by including trig and exponential functions.

Inclusion of the source code with Turbo Professional is a nice touch. If you think certain routines ought to work differently, you can change them. Some are written in Pascal, but most are in assembler, so you'll need MASM or something like it if you plan to do much rewriting. Of course you can also add functionality to suit.

I have only one complaint about Turbo Professional. I never did figure out how to work the Menu Maker utility, and the documentation didn't help. That problem aside, for serious developers who work in Turbo Pascal 4.0, Turbo Professional is an outstanding productivity aid that will pay for itself many times over.

by Kent Porter

Brief

Product:
Brief, Version 2.01
Target:
PC or PS/2 and compatibles
Requires:
DOS 2.0 or later; 192K minimum
Pricing:
$195
Vendor:
Solution Systems
541 Main St., Ste. 410
South Weymouth, MA 02190
(617) 337-6963

One might fairly describe Solution Systems' Brief as the quintessential programmer's editor. Unfortunately, that doesn't tell you much. So maybe I'd better tell you why I think it's so terrific.

To begin with, it's chock full of the features that programmers routinely require. And in the event that you find it wanting in a specific function or feature, Brief has its own C-like language for writing macros or otherwise tailoring it to your requirements.

Although Brief isn't copy protected, it's necessary to run an installation program to configure it. There are scads of options: colors, cursor shapes, tab settings, 25– versus 43– versus 50–line mode, backup files, and what not. This process uncovers some intriguing features. For example, you can set up Brief to save your work automatically every so many seconds or minutes, thus minimizing the impact of a power failure. For another, you can establish different parameters for each language you use.

Brief does all the usual editing things: line insert/delete, search and replace, block move/copy, etc. Most editing commands are accomplished by ALT-key sequences.

Brief gives you plenty of room to work in; maximum size of a text buffer is 32 Mbyte (DOS limit), with the maximum number of buffers limited only by available RAM. Macro size is also plentiful with a 32K limit and the maximum number of macros limited by available RAM.

A particularly nice feature is the Undo command, ALT-U. Since Brief maintains an internal change stack, by default, this stack keeps track of the most recent 30 changes. And if you wish, you can increase it as high as 300. So, if you find that some program modification is headed down the primrose path, just start hitting ALT-U to back out the changes you've made. Brief "undoes" them in LIFO order.

The real stuff of Brief is its multifile and windowing capabilities. The program comes up in full-screen edit mode, where one file owns the entire screen. You can load any number of files and move among them either in carousel fashion or via direct jumps using ALT commands, with each in full-screen mode. Alternatively, you can split the screen into resizable tiled windows and put a different file in each one, or different parts of the same file in two or more windows. If several windows contain the same source code, a change in one immediately appears in all.

The import of this is that if two modules are related, you can look at both on the same screen, jumping between to reconcile their differences. Or, in a single program, you can display a structure definition in one window while you write its initialization code in another, which displays a different part of the file. In a complex application, this feature alone can save an astronomical amount of time simply by eliminating the need to save and load files, scroll around, and print listings that will immediately become out of date.

Also, Brief's cut-and-paste and copy operations span multiple buffers using a clipboard concept. This lets you take a function out of PROG1 and insert it into PROG2 with half a dozen keystrokes.

On the negative side, moving among windows is awkward. You press the F1 key, then a cursor arrow indicating which direction you want to go. You have to repeat this in every window you pass through between origin and destination. It would have been better if they'd numbered the windows so you could do F1-window number.

Brief lets you set up and select language-specific environments. For example, indentation conventions are different for Assembler than for block-structured languages such as Pascal, C, and Modula-2, as is case sensitivity (Pascal and Assembler aren't, C and Modula are). The default filename extensions also differ. The setup program builds parameter sets for each language. In it you can also specify the compiler command line, enabling you to invoke the compiler from within Brief. This effects a Turbo/Quick-style "environment," although of course there isn't the same degree of interaction between editor and compiler that characterize true integrated environ-

(continued from page 138)

ments. Still, it saves time when you don't have to drop out of the editor to compile, then reenter it to fix errors.

I use Brief a lot, and aside from the window-changing problem mentioned earlier, there's only one thing about it that really annoys me. That is that you can't use the gray asterisk to type the multiplication symbol; you must use Shift-8. Brief regards the gray asterisk as a synonym for ALT-U (Undo). I should probably reconfigure the keyboard, which Brief lets you do, but I haven't gotten around to it.

Brief comes on two floppies and with two spiral-bound manuals in a slipcase: a 161-page editor reference and a 195-page macro language guide. Both contain comprehensive tutorials written in English. The 36-page editor tutorial gets you productive with this excellent programmer's tool in about an hour.

Like I said, it's terrific.

by Kent Porter

Guidelines C++

Product:
Guidelines C++, Version 1.2
Target:
PC and compatibles
Requires:
DOS 2.1 or later; 800K of hard disk space; 640K of RAM; and Microsoft C Version 3.0 or later
Pricing:
$295;
owners of previous versions $25
Vendor:
Guidelines, P.O. Box 749, DDJR
Orinda, CA 94563
(415) 254-9393 or (415) 254-9183

If you've been programming in C for a while, enjoyed structures and pointers to functions, and realized that there is something someone could do to structure the programming even more, you've likely already postulated a language like C++. C++ is still fairly new (sort of like C was in 1987), not terribly popular (yet), and extremely powerful.

C++ is an object-oriented language, which means, in short, that more emphasis is placed on the design of objects and object manipulators than on functions and procedures. Like C, C++ was deveoped by AT&T. Version 1.2 is their third release following Version 1.0 and 1.1. Guidelines' C++ for MS-DOS Version 1.2 is ported from the AT&T release of the same number.

It's important to note that the C++ translator is not a compiler, but rather works in conjunction with one. A C++ program is first preprocessed, then translated to C, then compiled by a C compiler. C++ is, though not strictly, a superset of C and most C programs will flow through the preprocessor and translator unchanged. In fact, some of the extensions to C that the ANSI committee has adopted were introduced in C++.

I found this package easy to install and flexible enough to deal with such things as alternate drives and directories than those recommended. I would have liked the option to install the package in the same directories as the C compiler it worked with, but it's easy to rearrange the directories later.

Several subdirectories are created within the main program directory. The LIB directory holds the libraries, one each for the five standard memory models. The INCLUDE directory contains all of the AT&T supplied headers, modified only enough to avoid conflict with the Microsoft headers. The SZAL directory holds information for each memory model about the size and alignment of different sized objects, none of which you should ever need worry about. The SCR directory contains a directory for each of the chapters in the Stroustrup book, plus the source code for a batch file builder. The DOC directory contains documentation on the C++ preprocessor and the C++ translator, as well as a copy of the tutorial chapter of the manual.

The last directory created is the BIN directory which contains the .EXE files for the C++ preprocessor and translator, as well as the batch file builder. Also in the BIN are 24 batch files (built with batch file builder) that take .CPP files to .CI (preprocessing done), to .CC (translator output), to .OBJ (compilation complete), or to .EXE for each of the five memory models plus a default.

Batch files are provided to allow the translator more working memory. Of course, MAKE could be used to invoke all of the appropriate programs individually, but more memory is absorbed by the MAKE utility than by the built-in batch file processor.

The Bjarne Stroustrup book, *The C++ Programming Language*, is included but may be omitted to save $10. The AT&T release notes are also provided which document differences between C and C++ as well as the differences between 1.0, 1.1, and 1.2 of C++. An installation guide and tutorial (really a series of examples) round out the documentation.

Free support is by mail only, unless you pay $50 per year to have access to telephone technical support. I'd prefer that they allowed some sort of phone access for a short period after acquisition (NOT registration). On the other hand, $50 doesn't seem unreasonable, and in the worst case, you can think of the price of the package as $345 and purchase the first year at the same time as the package.

Finally, from my contact with them, Guidelines seemed like reasonable people and I suspect that they would allow some simple questions to be answered without charge. At least they answered the questions I had during installation. I found few glitches in the package, none serious, just annoying. I liked the fact that all of the chapter directories included a makefile, however, the makefile .SUFFIXES command was not quite complete lacking the .EXE extension. This "bug" has been reported and will likely be fixed long before you get yours. Overall, I liked the package, and given sufficient free time to learn yet another language, I'll use it regularly.

by Richard Relph

LETTERS
(continued from page 14)

For example, I prepared the minimal dynalink module shown in Example One, and compiled it with IBM C/2 using the options *–c –Gs –-ALu* which, literally translated, mean *compile only, no stack checks, large model with no dependency on DS*. Then I tried to link the object module with the definition file

```
LIBRARY APROC
EXPORTS aproc
```

which requests linking a dynamic link library. The link reported an unresolved external of *___acrtused*, which I recognized as an entry point into the C setup routines. When I supplied the C runtime library, the link pulled in all of the C setup code and generated an unresolved extern for *__main!* I circumvented this by adding the line

```
void __acrtused( ){ }
```

to my module. This satisfied the external reference, and in fact the generated code did not actually call the dummy *___acrtused* function.

Thus finagled, the module did link and would have worked as a dynamic link package. But when I changed it to use a floating point number, as shown in Example Two, the link failed with many unresolved externs in the float libraries. These float routines are known not to be reentrant, so they can't be used in dynamic link code. Neither can the C library routines for file I/O.

In short, it is possible to build dynamic link packages using the official OS/2 C compilers. You have to know the ins and outs of the compiler very well, restrict the code to integer arithmetic, and do all I/O by way of calls on OS/2 system functions instead of the C library (this last is no great hardship, as the OS/2 system functions are quite complete). With other languages and compilers, good luck!

DDJ

```
int far pascal aproc(char far *arg1)
{
char locv[8];
        locv[2] = *arg1;
        return(locv[2]);
}
```

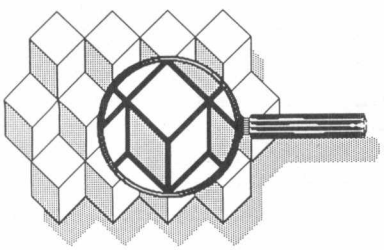
Example 1

```
void _acrtused(){}
static double bignum = 0.0;
int far pascal aproc(char far *arg1)
{
        bignum += *arg1;
        return(bignum);
}
```

Example 2

PROGRAMMER'S SERVICES

OF INTEREST



Languages

Coral Software Corp. and **Franz Inc.**, in a combined development effort, have announced Allegro CL, a complete implementation of Common LISP for the Mac II. The product features an incremental native code compiler, object-oriented programming capabilities, a programmable EMACS-style editor, and advanced debugging tools such as a window-based inspector and stepper. These are fully integrated into the Macintosh user interface to create an unparalleled LISP programming environment. Thus, Allegro CL allows the Mac to function as an independent AI workstation.

Allegro CL requires 1-Mbyte RAM and 1.6 Mbyte of disk storage. It runs on both 68000 and 68020 CPUs, directly supports the 68881 math coprocessor, and runs on Macintoshes with large screens. The product sells for $399.95. Reader Service No. 24.
Coral Software
P.O. Box 307
Cambridge, MA 02142
(617) 547-2662

The Golden Common LISP (GCLISP) Developer 286, Version 3.0 is available from **Gold Hill Computers**. The new version features a compiler which strictly adheres to the complete semantics of Common LISP, faster load times, an expanded GMACS editor for easier manipulation of LISP syntax, and support for Portable Common Loops, a flexible object programming system. The GCLISP Developer also comes with the San Marco LISP Explorer, an interactive tutorial developed by San Marco Assoc.; the GCLISP Reference Manual; User's Guide; the second edition of LISP by Winston and Horn; and the Common LISP Reference Manual by Guy Steele. Reader Service No. 25.
Gold Hill Computers
163 Harvard St.
Cambridge, MA 02139
(617) 492-2071

PowerLisp *Version 2.0*, from **Micro-Products,** offers complete support for Common LISP and InterLISP-10 as well as Common LOOPS. PowerLisp supports the complete Common LISP standard. The product is available for the IBM PC AT, Intel 386, and VAXmate. PowerLisp for the 286 is $2,195 and for the 386 is

$2,995. Reader Service No. 26.
MicroProducts
370 W. Camino Gardens Blvd.
Boca Raton, FL 33432
(800) 553-0777

Digitalk is now shipping release 2.0 of Smalltalk/V, a PC-based implementation of the Smalltalk programming language. The new release supports the high-resolution graphic modes (640 × 480) of the IBM PS-2/25 and 30 computers and includes windowing enhancements.

Smalltalk/V Release 2.0 is priced at $99.95. Registered owners of earlier releases may obtain upgrades for $25. Reader Service No. 27.
Digitalk
9841 Airport Blvd.
Los Angeles, CA 90045
(213) 645-1082

Trilogy is a new programming language developed by **Complete Logic Systems** that combines procedural programming, declarative programming, and database programming. Trilogy comes with its own environment including editor, module library, interactive compiler (producing native 8086 and 8087 code), online linker, loader, and context sensitive help screens. Users can edit, compile, link, and execute multimodule programs without ever leaving the Trilogy environment. The product also comes with four standard modules called Math, Strings, Files, and Windows. These modules export routines (in that order) for transcendental functions, string/date/time manipulation functions, file access functions, and windowing functions.

Trilogy runs on the IBM PC, IBM PC AT and XT, and compatibles using DOS 2.1 or later, uses 512K, and is not copy protected. It is priced at $99.95 US. Reader Service No. 28.
Complete Logic Systems
741 Blueridge Ave.
North Vancouver, B.C.
Canada V7R 2J5
(604) 986-3234

Books

McGraw-Hill has published a new book called *A Comprehensive Guide to AI and Expert Systems: Turbo Pascal Edition* by Robert Levine, Diane E. Drang, and Barry Edelson. The book offers full explanations of the basic concepts of artificial intelligence and expert systems and also shows how AI techniques can be implemented on a personal computer. Complete discussions examine a wide range of important topics, including natural language processing, forward and backward chaining, and the use of probability and fuzzy logic in expert systems. Aspects such as object-oriented expert systems, semantic nets, certainty factors, automated learning, using PROLOG to design expert systems, and LISP are also explored. The book sells for $19.95. Reader Service No. 29.
McGraw-Hill Book Company
11 W. 19th St.
New York, NY 10011
(212) 337-5945

Structured Induction in Expert Systems, by Alen D. Shapiro, is a new book published by **Addison-Wesley** which shows how the knowledge acquisition process can be automated by the application of inductive learning techniques. It provides a guide to techniques for the synthesis of transparent rules from examples supplied by the domain expert. It also shows applications of these techniques to complex problems in chess end-game classification. This book can be used as a supplement for graduate-level courses on expert systems and AI programming. The suggested retail price for the book is $31.25. Reader Service No. 30.
Addison-Wesley Publishing
Reading, MA 01867
(617) 944-3700

Expert Systems

OPS83, by **Production Systems Technologies** is a rule-based programming language that was designed as a tool for developing and delivering knowledge-based, second generation expert systems. The language is not restricted to merely diagnostic or classification tasks. Code written in OPS83 can easily be interfaced to code written in other languages. Reader Service No. 31.
Production Systems Technologies
642 Gettysburg St.
Pittsburgh, PA 15206
(412) 362-3117

VP-Expert from **Paperback Software** is a rule-based expert system development tool. Features include a built-in text editor, an inductive front end that creates if-then rules directly from examples in external files of other programs, an inference engine, floating-point arithmetic and trigonometric functions, confidence factors, text and graphic tracing, and external program calls.

VP-Expert runs on the IBM PC, IBM PC AT and XT, IBM PS/2, or compatibles with 384K, uses MS-DOS 2.0 or later, and is not copy protected. The product sells for $124.95. Reader Service No. 32.
Paperback Software International
2830 Ninth St.
Berkeley, CA 94710
(415) 644-2116

The Berkshire Software Company has released Turbo Shell Version 2.0, a software system designed to provide a complete environment for the development of expert systems. The product provides menu-driven facilities for creating, modifying, and consulting knowledge bases comprised of rules, facts, and solutions. The system is based on the

classic MYCIN model of expert system development using production rules for the classification of knowledge. Turbo Shell can be requested to provide an explanation of its logic and line of reasoning in pursuit of a conclusion, and will provide a complete explanation when a conclusion has been reached.

The Turbo Shell expert system development environment is written entirely in Turbo PROLOG. The package sells for $89 and runs on IBM PCs and compatibles with one dual sided floppy-disk drive and at least 256K of RAM. Reader Service No. 33.
The Berkshire Software Company
44 Madison St.
Lynbrook, NY 11563
(516) 593-8019

Miscellaneous

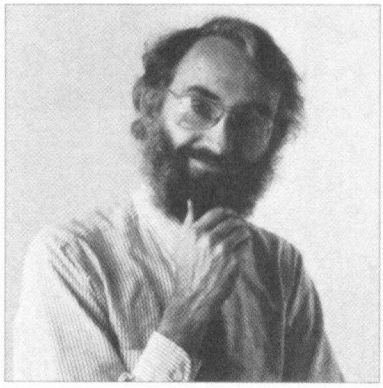
AI-NET 101 from **AI Ware** is a data analyzer based on neural net technology. It autonomously learns patterns among input samples using techniques inspired by knowledge of the human brain's learning processes. AI-NET 101 operates in real-time and is suitable for integration into industrial environments. Its modular design allows several nets to be connected into a single system. AI-NET 101 uses proprietary software that takes advantage of the neural net's internal structure to increase its learning performance. AI-NET 101 uses a Microsoft Windows interface. The AI-NET accelerator card, which fits into the IBM PC, increases learning speeds up to 10 times and more. The product sells for $4,500. Reader Service No. 34.
AI Ware Inc.
11000 Cedar Ave., Ste. 212
Cleveland, OH 44106
(216) 421-2380

CLOE is a new software package developed by **Symbolics** that is a software environment for both the Symbolics 3600 product line and Intel 80386-based personal computers. CLOE consists of three software modules—one for delivery and two for application development and fine tuning. For delivery, CLOE Runtime executes the application on the targeted 80386-based delivery machine. This software module is a native 80386 Common LISP runtime platform with extensions such as multitasking, support for new flavors object-oriented programming language, advanced exception handling facilities, and a high-performance garbage collector. For development, programmers will need the 3600-based Developer software module and the companion 80386-based Application Generator.

CLOE Runtime requires an Intel 80386-based PC or workstation, Unix System V, Release 3.0, 2 Mbyte of RAM and a 20-Mbyte hard disk. Prices vary according to licensing configuration. Reader Service No. 35.
Symbolics Inc.
11 Cambridge Center
Cambridge, MA 02142
(617) 621-3727

DDJ

SWAINE'S FLAMES



Whan that April with his showres soote The droughte of March hath perced to the roote, And bathed every veine in swich licour, Of which virtue engendred is the flowr... Thanne longen folk to goon on pilgrimages, And palmeres for to seeken straunge strondes....

—Geoffrey Chaucer

Language evolves.

John McCarthy invented the LISP programming language almost 30 years ago, describing its overall design in an April 1960 *Communications of the ACM* paper. Since that ancient April, LISP has evolved in response to new environments: supercomputers, time-sharing systems, personal workstations.

During those decades, LISP maintained a solid reputation for being powerful, elegant, and inefficient. Remarkably inefficient. LISP code gobbled memory voraciously. LISP programs ran slowly, bogged down in function calls. That was the perception.

The perception was mainly accurate. Early compilers for LISP were crude and produced "simple and often ridiculous code on backing out of an execution-order treewalk of the program," according to Richard Gabriel, who wrote a book on benchmarking LISP implementations (*Performance and Evaluation of LISP Systems*, MIT Press, 1985).

Today, though, LISP compilers use all the tricks of the optimizing compilers for algorithmic languages. They convert costly function calls to in-line code, delay evaluation and rearrange the order of evaluation for efficiency, unwind loops, and do peephole optimization and constant subexpression elimination and interfunction optimization. If you think of LISP and FORTRAN, two ancient languages, as the archetypal symbol-processing and number-crunching languages, respectively, you might be surprised to learn that one modern LISP implementation, S-1 LISP, produces code for numeric computations that rivals FORTRAN code.

A LISP program is still likely to run slower and to use memory less efficiently than a comparable C program. But increased memory and processing power in current hardware and continuing improvements in LISP implementations make LISP more appealing for inclusion in developers' toolkits. This is good news because LISP's virtues—its extensibility, its expressive power, its facility in handling symbolic information—are impressive.

It's time somebody said it: spelling checkers are snake oil. In a recent editorial in another programming magazine, the editor (let's call him David) congratulated himself on the acquisition of a spelling checker, which he touchingly expected to cure the misspellings that have plagued his editorials. The selfsame editorial was, of course, plagued with the kinds of misspellings that spelling checkers don't check.

Spelling checkers do not, in fact, check spelling at all, and anyone who uses one for that purpose has been suckered by the snake-oil salesmen. If you want to know how a word is spelled, David, you look it up in a dictionary. The word lists supplied with spelling checkers are not dictionaries, whatever their purveyors may tell you, and can at best tell you that there exists a word with a certain spelling. Spelling checkers won't stop you from putting that damned apostrophe in possessive its. The commonest errors are the ones they are most likely to ignore.

Spelling checkers should really be called typo catchers because all they do is catch certain kinds of typing errors. Unfortunately, they don't even catch all of these. Consider the following mish-mash, which any spelling checker would accept unquestioningly: "Now is the tome fir all god men to sump this sneak oil sown the grain." What kind of error checking is it that regularly fails to detect the most common errors? Sneak oil indeed.

As you read this, it will have been nearly six months since Rob Dickerson made his pilgrimage from the Pacific Northwest to the straunge strondes of Scotts Valley, California. After some wintry blustering, Microsoft and Borland came to an agreement that put certain short-term constraints on Dickerson's activities as vice president of product management for Borland and also included a pact that the two companies would not recruit employees from one another's ranks for six months. Next month, when the moratorium on raiding ends and Nature priketh them in thir corages, will other folk longen to goon?

Michael Swaine

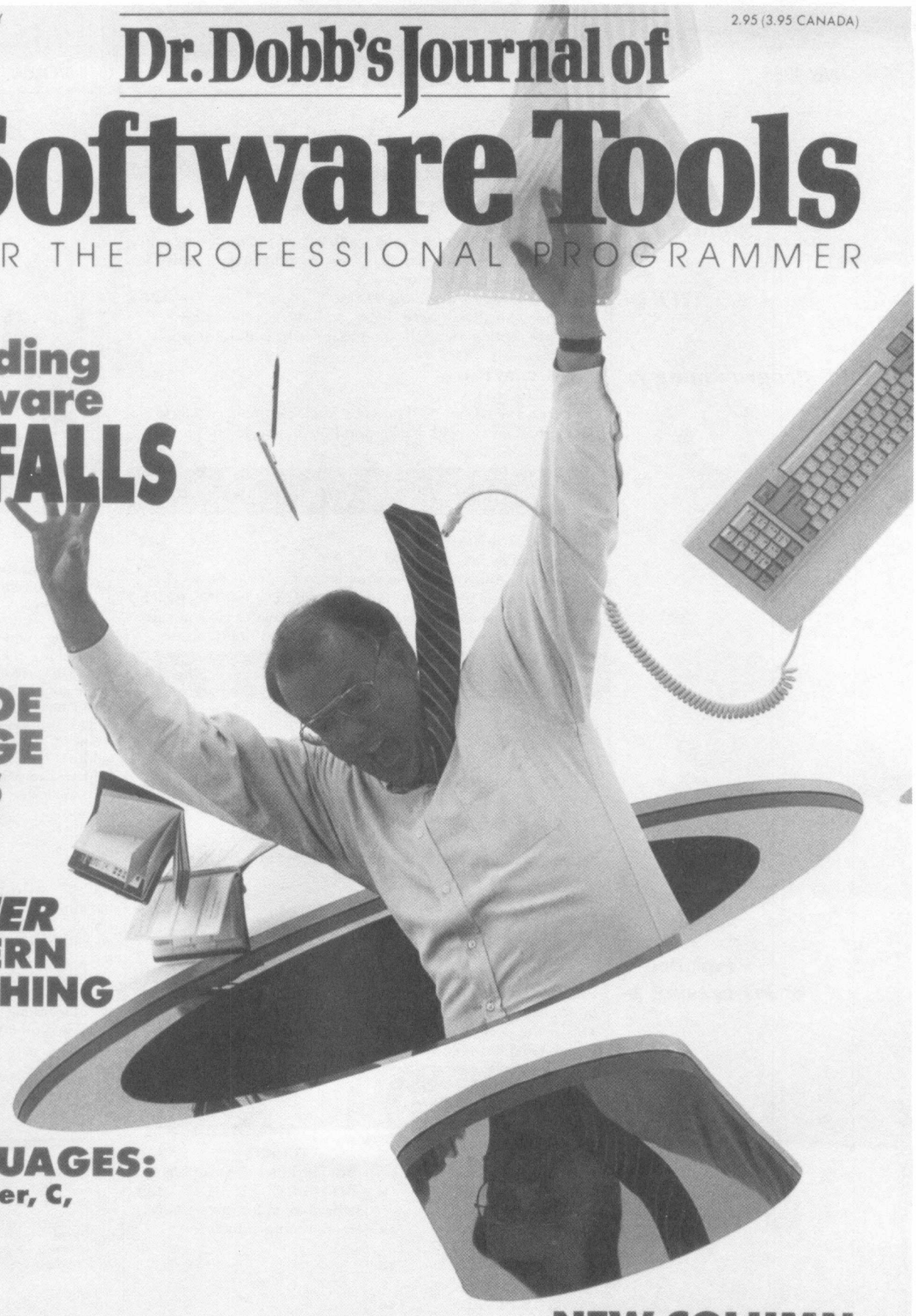
Michael Swaine
editor-in-chief

#139 MAY

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER



Avoiding Software PITFALLS

INSIDE IMAGE FILES

FASTER PATTERN MATCHING

LANGUAGES:
Assembler, C, Pascal

0 38351 16562 8 05

NEW COLUMN
Programming Paradigms

CONTENTS

ARTICLES

Designing Software ▶

Developing for the User 18
by Robert Carr
Developing good software is more than just programming and debugging. Carr's eight axioms of successful software delivery tell you the rest of the story.

TIFF ▶

Handling Image Files with TIFF 26
by Anthony Meadow, Rocky Offner, and Michael Budiansky
If you are writing an application that works with bitmapped images, consider TIFF. . . .

C Programming ▶

Virtual Array in C 46
by Mark Tichenor
Data arrays need not be limited by available memory. Mark uses virtual arrays with background file management to overcome this problem.

REVIEWS

EXAMINING ROOM 122
coordinated by Ron Copeland
Products examined from the programmer's perspective. This month's offering include PC/Forms tested in C, Soft-ICE, which provides the capabilities of an in-circuit emulator via software, and DE, a "stretched" version of the standard EMAC editor.

COLUMNS

C CHEST 72
by Allen Holub
Allen discusses the benefits of compiling to an intermediate language (postfix)—developing for multiple hardware platforms, merging compiler output, and eliminating redundant code.

TO THE MACS 82
by Stan Krute
Stan takes a field trip to the recent MacWorld Expo. He also has a wrap-up of the Scouting Toolkit.

Pascal ▶

STRUCTURED PROGRAMMING 92
by Kent Porter
With the help of a special model file—which Kent developed specifically for this exercise—he will demonstrate how to write a Pascal program that reads non-Pascal files.

Parallel Processing ▶

PROGRAMMING PARADIGMS 100
by Michael Swaine
This is the premier of a new column: Parallel Processing, Object-Oriented Programming, and a reading list. Enjoy!

FORUM

EDITORIAL 6
by Jonathan Erickson

RUNNING LIGHT 8
by Tyler Sperry

LETTERS 12
by you

CARTOON 14
by Joe Sikoryak

SWAINE'S FLAMES 144
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST 136
brief product descriptions

AD INDEX 137
where to go for more information on products

About the Cover

A successful conclusion to any journey means that you have to know where you're going, including which pitfalls to avoid along the way if you're going to arrive safely. These same conditions apply to the goal of developing software that is both well designed and well received. Here's hoping this month's modest contributions help you to avoid those nasty pits of alligators scattered along the way.

Next Issue

In June, C-columnist Allen Holub stalks the wild memory allocator, Mike Swaine reports on a recent gathering of programming prognosticators, and, in general, DDJ examines real-time operating systems, dedicated and otherwise.

EDITORIAL

Four years ago this month, Mike Swaine wrote his first editorial for *Dr. Dobb's Journal* and in it he talked about *change*—changes at the magazine, changes in the microcomputer industry, changes in general. As it turns out, *change* is the topic of this month's editorial too. You may have in fact already noticed one change. Mike has a new role with the magazine, having moved from the position of editor-in-chief to that of editor-at-large. This gives him a better opportunity to do more original writing, something he's wanted to do for a long time.

For me, Mike's change means a new opportunity. I am Jon Erickson, a former senior editor at *BYTE* and now editor-in-chief of this magazine. *DDJ* has long been one of the microcomputer magazines I've most admired and the chance to lead it into the coming years was a challenge I didn't have to think twice about accepting.

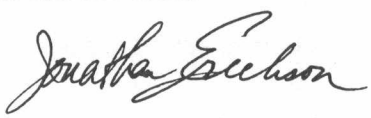
Mike's move doesn't mean that *DDJ* is going to become a radically different magazine from what it has been. I'm as committed to *DDJ*'s original spirit as Mike, and any changes that do take place will be dictated by technological advances and other significant trends that are important to you.

Just coincidentally, the changes at *DDJ*'s parallels some interesting shifts in the microcomputer industry itself. Advances in small system hardware architecture are making life miserable for manufacturers of expensive minicomputers and the emergence (finally) of more sophisticated operating systems like OS/2 promisespowerful new applications. Over the coming months, *DDJ* will be paying particular attention to how those changes affect the software development process.

This changing of the guard, so to speak, provides you with the opportunity to have a say in what directions *DDJ* will go in the future. Take a few minutes to drop me a letter about what you think *DDJ* should be doing and where it should be going. What are we doing right or wrong? What could we be doing better? What aren't we covering that we should be? If you have a specific article in mind that you'd like to write, let me know about it too.

One change I wouldn't mind seeing in the microcomputer industry is a shift away from the tendency of companies to pour resources into litigation instead of R&D. First it was the Lotus look-and-feel suit and more recently Apple's copyright infringement suit against Microsoft and Hewlett-Packard.

There's little any of us can do when influential companies decide to fight it out in the courts instead of on store shelves. But in the meantime, a lot of small (and large) software developers are caught in the middle, wondering whether or not they should sink more resources into further development of Windows applications. In truth, there's probably some merit to the suggestion that part of Apple's strategy is to delay and stifle the development of graphical user interface Windows-based applications. If so, then third-party developers should resent being manipulated like pawns on a corporate chessboard. Maybe by the time this editorial appears, the dispute will be resolved but I don't think so. In the meantime, it looks like it will be the lawyers who profit at the expense of independent developers and end users.



Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editor *Rhoda Simmons*
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Fulfillment Coordinator *Francesca Martin*
Subscription Supervisor *Kathleen Shay*
Newsstand Coordinator *Sarah Frisbie*

Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 137*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, with two special issues per year by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA 800-321-3333; in CA 619-485-9623 or 566-6947. For book/software order problems call 415-366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

The original theme for this issue was "Designing Applications," and while I still think it's a perfectly reasonable theme I must confess to some surprise at the variety of responses we got from authors. Space limitations forced us to trim the number of articles more than we would have liked, but that's a chronic problem with magazines and is certainly nothing to distinguish this issue. Suffice to say that we are not finished with the topic of designing applications software with this month's articles.

Something that does distinguish this month's issue is Robert Carr's article on guidelines for developers. After reading the article, you might be tempted to dismiss the message—developing for the user, after all, is something that everyone claims to do. Despite the apparent simplicity and familiarity of Carr's advice, I strongly urge you to read the article and then reconsider your development efforts. A careful examination of your current process might reveal a few holes.

It's tempting to dismiss another reminder of the importance of the user in software design. I might even agree (on a good day) that we all learned that lesson years ago, except for the overwhelming evidence to the contrary.

Item: Just a few months ago, a major software house was in the final beta pass for a product with an integrated editor. It was only at this last pass that someone at a beta site complained that the editor was abysmally slow. The reviewer's comments—something about the seasons moving faster than the cursor—caught the developers by surprise: the program had always seemed fast enough on their 386 machines. Alas, the target audience consisted primarily of people using 8088 machines.

Nor are these problems limited to performance issues. A year and a half ago I pretty much trashed Zoomracks in a review that appeared (substantially rewritten) in *BYTE*. The keyboard interface was a Byzantine complex of Alt- and Control-key commands, the mouse interface—what there was of it—was buggy as hell, and the file facilities could be charitably described as primitive. (The PC version, for example, couldn't recognize subdirectories.)

The capper to this second story is that Paul Heckel, the designer of Zoomracks, is also the author of *The Elements of Friendly Software Design* which is a pretty good book on designing applications.

These aren't isolated instances. There isn't a month that goes by that I don't examine a new program that seems hastily thrown together. Little touches like using subdirectories and environmental variables really aren't that hard to add. And besides, we all know better. End of sermon.

This month's news is probably already old to you: Jon Erickson, our new editor-in-chief, is online starting this month. You can meet him back on page 6.

This month's reading list is a short one. Go to the local university library and pull out George A. Miller's classic 1956 paper, "The Magical Number Seven, Plus or Minus Two." Then consider the implications of human bandwidth on your next design. A little thought might make a world of difference for your customers.

Tyler Sperry

Tyler Sperry
editor

ARCHIVES

Parsimony

"A software development project attempts to solve a problem by producing a solution which includes a software system. Any acceptable solution must meet a set of functional requirements and a set of constraints. Constraints are negative requirements (e.g., compatibility, performance, interface specifications). The more requirements a project has, the more difficult it will be to meet them all.

A particular problem has some fixed requirements and many flexible ones (i.e., unspecified or loosely specified). When a system gets designed for execution on a computer, additional constraints must be imposed (e.g., sequential execution, control structures). Additional constraints are imposed when it is implemented (e.g., finite resources, discrete arithmetic). These constraints are 'artificial' in that they did not come with the original problem.

It is important not to impose any more artificial constraints than necessary in order to aid productivity. Then more room is left for trade-offs which can produce a more desirable result. These views come directly from the belief that simplicity is better than complexity. . . ."—*Kim Harris, "The FORTH Philosophy,"* DDJ, *September 1981.*

Syntactic Sculpting

"A hacker is an artist, and computer artistry is not distinguished from other art forms except in the medium chosen. . . .

"It is the nature of great art that it compresses a great deal of design, intellectual sweat and individual personality into the created object. There are no great paintings painted by a committee or artists. Curiously, it seems to matter little what tools the artist had, or where his starting point was, provided that his accomplishment from that point exceeds what the rest of us could have done. Ansel Adams had color and motion available, but he chose to limit himself to black and white stills—and what art he created! A virtuoso on a violin produces art; the same sound from a Moog is ho-hum. . . .

". . .Unlike painting, sculpture, music, and the like, few people can really appreciate the artistry in a computer product. Perhaps that will change."—*Tom Pittman, "Festschrift for Doctor Dobb,"* DDJ, *February 1985.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

MASM Complaints

Dear *DDJ*,

It's amazing to see the phrase "screamingly fast" applied to the lumbering leviathan that is MASM 5.0 (Examining Room, February 1988). Compare the reviewer's measurement of 120 instructions per second to the greater than 1,000 instructions per second of Eric Isaacson's A86, which assembles directly to executable code and simultaneously creates a symbol file and embeds error messages in the source code. And it does all this quicker than MASM links. I used it to assemble a 12-module program to a .COM file in 4.8 seconds; it took MASM 54 seconds to assemble the same program and 5.7 seconds to link it.

The real purpose of this letter is to complain that MASM 5.0 no longer supports .COM files and nobody says anything about it. CodeView does not offer symbolic debugging of .COM files and MASM 5.0's symbol file is not compatible with Symdeb or other symbolic disassemblers. So even if you're satisfied with the creepy-crawly rate at which MASM 5.0 assembles and links, you have to give up .COM files or symbolic debugging.

I wish you would have a person review assembly-language products who does not think that assembly language is the "most tedious of programming languages" and that "the emphasis is shifting away from assembler as a primary language" and does not call assembly language "assembler." A person who can't be bothered to distinguish verbally between the language and the translator and who thinks that assembly language is unpleasant will not pay attention to details important to assembly-language programmers.

I ask you: In what other language can you write an assembler that assembles 1,000 instructions a second while it does other things and ends up occupying 20K? Or in what other language can you write a utility that searches for a file name in a full 10-Mbyte, 70-subdirectory hard disk in 12 seconds? Why do advocates of one tool (a higher-level language in this case) disparage others?

George Frank
Encinitas, Calif.

Kent Porter responds:

In comparison with earlier releases, MASM 5.0 is "screamingly" faster. There are even faster assemblers, and I'll take George's word that A86 is one of them. It's true that MASM 5.0 doesn't produce .COM files directly; you have to run the end product through EXE2BIN if you want a .COM. It's also true that I didn't mention that, and I apologize on behalf of all us reviewers who haven't said anything about it.

Assembly language is indeed tedious. That doesn't mean it's bad, nor did I suggest that it is. It's a tool and it has its place. Most developers have switched to C, Pascal, or Modula-2 because they're more productive (that is, less tedious) languages, writing only high-overhead routines in assembly language. Thanks, George, for setting us straight on the proper use of terminology. When I started programming IBM mainframes in the early 60s, we called the language "assembler," and I think I overheard some other professional programmers still misusing that same term just last week.

It's a revelation to learn that writing about structured programming makes me an "advocate" in some grandiose struggle for language supremacy. Let's try to keep things in perspective, shall we?

Don't Believe Everything You Read

Dear *DDJ*,

I am embarrassed to say that I must retract two statements I made in my letter published in the November 1987 issue of *DDJ* on the subject of teletypewriter terminals. I was careful to get my facts concerning teletypewriters correct, but I was not as careful with the examples I gave of other standards that had outlived the reasons for the standards being set the way they were. I received a letter from Mr. Clive J. Grant, a professional engineer, debunking both of these examples.

I have forgotten where I read the story of the Roman emperor and the railroad gauge, but it was interesting and plausible and I fell for it. According to this tale, the Roman military had a problem with ruts. Ruts in unpaved roads tend to enforce at least a local standard on wheel spacings because a vehicle with a wheel spacing not matching the rut spacing is in difficulty. The trouble was that in different parts of the Roman Empire, there were different local standards, and this created problems with chariots and other military vehicles when the legions were moved from one place to another in the empire. Therefore, the emperor issued a decree standardizing wheel spacings in the empire, and this standard, enforced by the ruts, endured long after the empire had fallen.

When the railroads were first started, so the story went, the developers turned to the carriage makers for the rolling stock, and this resulted in the standard being transferred to rail spacing. In fact, when Stephenson established the railroad gauge, rail systems were in use in the Cornwall mines and he took an average of the mine rail spacings, which varied widely. There is some evidence that there was a Roman standard for wheel spacing, but it may have applied only to Rome and the immediate vicinity and in any event had no connection with the railroads.

The typewriter keyboard tale came from an article advocating the new keyboard layout that is supposed to be much faster. Mr. Grant points out, however, that Sholes, who originated the QWERTY key-

board, never revealed why he arranged the keys in that manner, so any reason advanced for it is speculation. Further examination of this particular speculation shows that its motivation was not to slow the typist, but rather to reduce key pileups.

On the old mechanical typewriters, if a new key was struck before the previous keystroke had retracted, then a pileup occurred. Therefore maximum typing speed was limited by the time required for the spring to retract the previous keystroke after the key had been released. However, many pileups occurred when a new key was struck with a different finger before the previous key had been released. The speculation was that the keyboard was arranged to assign groups of keys apt to be struck in succession to the same finger and thus ensure that one key would be released before the next was struck. This is not quite the same as trying to slow typing.

I hope that my blind acceptance of interesting stories I have read has not caused too much of a problem by misleading your readers.

David S. Tilton
Manchester, N. H.

HyperCard Ends an Era?

Dear *DDJ*,

I appreciated Mike Swaine's comments concerning HyperCard in "Running Light" in the January issue of *DDJ*. Many of his points were right on the mark. Others, however, were wild shots that may sound logical but not to an ol' Mac end-user. HyperCard will indeed bring about a proliferation of stackware, and no doubt there is going to be a lot of sloppy programming. This, I agree, is inevitable. Will it threaten the Mac user interface as you suggest? This I seriously doubt.

HyperCard is merely the growing momentum in computerdom to making the world of computers more "user friendly." Stackware may indeed become polluted for awhile, as everyone with a new Mac jumps into the programming ring. But when all the smoke finally clears, what you will have is the availability of narrowly focused programs that serve a limited consumer base that would not otherwise be met by commercial programmers. The reason? Lack of interest, lack of monetary reward, and most of all lack of knowledge in or about these highly specialized areas.

The authors of many of these stacks will be professional people like myself who have an interest in a particular area and are acutely aware of special needs. Our reward for meeting those needs will transcend monetary gains. We are by nature not "sloppy," particularly in our work or whatever interests us. It doesn't mean we will challenge Microsoft or even rival Danny Goodman's work, but it will be good. Above all it will meet the needs of small groups of people who would otherwise be ignored.

I don't believe HyperCard spells the end of anything. Just as the Macintosh has made computers easier "for the rest of us," so will programs such as HyperCard make producing programs easier. That is as it should be. The progress that has been made in making computers easier to use in both the hardware and software will not only continue but will also accelerate.

Programming will not be immune from this progress. We will see "home videos" in software, but Microsoft or even your local computer store won't be selling them—any more than NBC or your local T.V. channel shows home videos. There will be a lot of amateurish stuff around, but it won't hurt anything. Indeed, it will help. It will stimulate imagination and generate interest. There will be a lot of really good stuff that you will never hear of because of limited interest and distribution. No, HyperCard isn't the beginning of the end, or even the real beginning—that occurred with BASIC or perhaps even before. It is just one more milestone in the evolution of the information era.

Ronald L. Cox
Poplar Bluff, Mont.

" 'Faster running' was last year's marketing plan. We wanted to sell something with a BANG!"

(continued on page 132)

Developing for the User

by Robert Carr

How do bad software products come about? The reasons vary, but one of the most common faults is when programmers design the product "for themselves," forgetting the product's real users. The key to avoiding this common pitfall is to realize that programming is ultimately for the user—the consumer of the software. You must view the process of developing software as a user-driven, software development cycle.

Software development is much more than programming and debugging; it's an overall life cycle of delivering software products to users. So central is the relationship between the user and the software product that the overall process should be called software delivery, not software development, emphasizing the delivery aspect as the clear goal.

The cycle of software delivery can be broken up into several phases, each with its own axioms. This article lays out eight such phases and their dictums. From the viewpoint of user-driven development, each phase offers a special contribution to the goal of successfully delivering software that makes your users rave about, demand more of, and pay for your software (and ultimately your salary). Although the eight phases are listed here in their natural order, in reality they often overlap; ideally, there is quite a bit of iteration and looping.

A last prefatory comment—although I refer to project teams, all these comments apply equally to the solo implementation effort.

Know Your Users

Just as journalists and writers for decades have offered the sage advice to "know your audience," so should developers know the users of their products.

How can you find your users, so as to know them? Some developers are hired directly (say, as contract programmers or through corporate MIS departments) by their users or users' organizations that will use their products; these developers are fortunate—they know exactly where to find their users. Other developers in some sense develop for "markets" of unknown users; they must find a representative sample of potential users.

Once you've found your users, do something simple with them: talk to them yourself. Talk to

Robert Carr is vice president of software for GO Corp., 139 Townsend St., San Francisco, CA 94107. He is the creator of Framework II and former chief scientist at Ashton-Tate.

them extensively, and do it before designing your product. If you have the sometimes mixed blessing of having a formal marketing department coming between you and your market/users, insist on accompanying or bypassing the product-marketing people to talk directly to users.

Much useful information can only be gained by direct contact with users. Software that makes users happy is hard to deliver on time and within costs and can only be created when design decisions are made with the most accurate projection of the users' reactions as is possible.

Although this may sound surprisingly simple, talking with users yields tremendous benefits when compared with the costs of presenting your users with formal marketing surveys or detailed product mock-ups.

Discuss the proposed software with them. Have they used similar packages in the past? What did they like and dislike about them? What are their hardware and software environment requirements? When would they ideally like the software by; when do they absolutely need it by? Or, if you're developing for the consumer market, what marketing windows do you have?—for example, when do you have to deliver in order to beat competitors to market?

One of the most powerful tools you have for meeting a targeted ship date is your ability to adjust the functionality and scope of your first version to fit the time available. Early, extensive user dialogue provides you with a gestalt feel for your market so you can make design trade-offs intelligently.

Select Appropriate Tools

Craftsmen need sharp tools. Happily, because personal computers represent the first mass market ever for programmer tools, they're beginning to host the best programming tools ever. Unless you are in the business of selling tools, avoid building them yourself unless absolutely necessary—your job is building the product for the users, not tools for yourself. It's likely these days that someone else has already built the tool you need and is willing to sell you a copy for a few hundred dollars. Buy it and profit.

As with tools, so with building blocks. Seek to partition your architecture such that as many pieces as possible can be subject to the "buy vs. build" decision. Strive to prejudice yourself toward buying. And even when the final version shouldn't be shipped with a preexisting building block, consider using it in the early phases of your project so as to get the overall system running as early as possible. This is a vital technique, supporting the evolutionary-development technique discussed later.

Spend Time on Preliminary Design

Now that you understand the needs of your users, don't rush into programming; instead, spend adequate time designing an architecture that meets all major product requirements, including user interface and performance. Go this far, but by all means do not go too far and extend this design phase until all aspects are finely designed to the smallest details!

Your preliminary design should give you a good handle on key algorithms, data structures, user-interface metaphors, major menus and their rough contents, the number and role of the internal modules, and the interfaces between key architectural layers.

At this point do not continue through detailed design of all prompts and strings, all menus and commands, all dialog boxes, all module interfaces, or writing of the entire program in pseudocode. Although you will need to do many of these things in the course of the project, by deferring detail you gain flexibility to change the product based on what you learn as you build it.

Although much of software engineering research and theory stems from the requirements of large military and aerospace software projects, we in the PC business need to be wary of applying these techniques to our work. What's necessary to produce satellite-control software at a large company is not necessarily the appropriate software development paradigm for us. And nowhere is this more true than in the need to avoid committing the complete design to paper before ever beginning to build.

Now is the time to perform selected tests, especially to highlight major performance issues—run a test to prove your design is sufficiently efficient. If, for example, you wonder if your windowing environment can paint characters fast enough, then write a test program now. If it doesn't, you know you have to redesign your display algorithms to minimize character painting.

The reason behind spending adequate time on design is to put your best efforts into avoiding major surprises or gotchas in the overall design of the product; these are the most costly to rectify. Even with your best efforts, it's likely there will be a couple of medium-size changes of direction/design that only become apparent in mid-course.

Use Your Team's Latent Talent

The most underutilized resources on most development projects are the programmers themselves! Often they are not allowed to discuss their own ideas with users or marketeers. Often they are pigeonholed into their one area and aren't even allowed to contribute to areas outside their own.

Ask discipline, professional work habits, and friendly interpersonal skills of your development team, and couch design and planning sessions in a consensus atmosphere that invites the best from everyone. Your team will then take the project closer to perfection than you guessed was possible. Emphasize the word *team*: what the team should value and aim for is what they can do together, not what any of them achieves individually.

A team-oriented consensus style is in strong contrast to concepts such as chief-programmer-based teams, for example, in which team members exist merely to make the guru chief programmer more efficient, implementing the designs that spring from his head. A benefit of inviting more contributions from team members is that their professional skills will develop more rapidly.

(continued from page 20)

Design and Develop Through Evolution

Once you have a clear understanding of your users' needs and your product design, you're ready to begin programming. It's key now to adapt an evolutionary implementation method that provides you with the flexibility to make changes as you learn from the growing product.

Some day there may be CASE tools that allow an entire product to be changed with ease, but for now the best way to obtain flexibility is to build from the outside in: begin building from the user interface inward and from your low-level building blocks upward. It's the middle levels of most software products that consume your implementation effort and are the hardest to change. For it is at these levels that functionality and behavior tend to be hard-wired into the architecture.

Low-level building blocks tend, on the other hand, to be understandable and flexible no matter what the flow and functionality of the higher levels. At the user-interface level, use scaffolding and other temporary crutches to get your user interface running. By doing so users can interact with the actual product itself, providing valuable polish and validation to your design.

If necessary you can prototype the user interface using prototyping tools such as Dan Bricklin's Demo program. Strive, however, to avoid using such tools for extensive mock-ups when you could spend your time developing the real code (with a little throwaway scaffolding).

Empathize with Users

If ease of learning is an important goal of your project, now is the time to sit a few users down and have them try to learn the system. Watch them yourself and have your programmers do the same (videotaping is a good method here). If three out of four subjects stumble at the same point, it's likely that area needs redesign. Such user testing can be both cheap and easy to perform. (If your market includes computer novices, for example, you can use relatives and friends as test subjects.)

A long-term benefit of having programmers and program designers talk with users and view user testing is that over time the programmers sharpen their "user empathy" skills. Over the course of a couple of projects, the programmers will develop user empathy—the ability to anticipate, during the design phase, many user problems your team will begin to get a reputation for being sharp user-interface designers.

Many programmers resist changes to their initial designs suggested by the marketing department or remote designers. But programmers who experience first-hand other humans having real difficulties tend to respond empathetically and are more motivated to iterate and perfect and polish the overall product for these users—all because their users are real to them.

Deliver the Product

It sounds obvious, but a far too common sin is not shipping the first version soon enough. In fact, some teams never ship "Version 1" but change and hack and perfect the product so long that they ship what can only be called Version 2.

Although sometimes justified by changing market conditions, waiting so long is usually a mistake. By shipping late you may miss a key market window or make your customers unhappy because they had to make do. If you'd shipped when you had a reasonable first version together, you may have released the product before any competitors did.

Support your product once it's shipping. Innovative products in particular need repeated explanation and selling from their designers. Ideally, your talks with users before ever building the product gave you a warm-up on explaining its special benefits.

Listen to Your Market

Despite your fervor in user testing and talking to users, there's no better way to get terrific feedback and guidance on evolving your product than shipping your product and then listening to the market.

Listen to your market by talking to the actual users of your product. Try to find a representative cross-sample of 15 to 25 users who you can talk to in the months after shipping. Question both beginning and advanced users of your product to get their reactions, but be sure to interrogate them again after a few months, when they've had time to plumb all the depths of your product.

Avoid the common mistake of releasing your software too often. Adequate testing and quality assurance at the end of a development project is expensive and often consumes four months. If you release a new version every ten months, you're testing four months for every six of developing. Furthermore, you have hardly any time to incorporate the expert users' feedback from your previous version. It's much better to adopt a 15-18-month cycle of releases, in which you have a full year or so of development for each testing phase. By adopting a discipline of following efficiently spaced releases, you can pull clearly ahead of your competition in features and functionality in just a couple of release cycles.

Reset to Phase One

The software delivery cycle is ideally one that loops repeatedly: Each of the eight phases is applied to every major product release, although some, such as selecting tools and building blocks, are obviously much abbreviated on later releases.

Developing for users is not easy. But it provides a methodology that helps you avoid costly errors that are surprisingly common: developing products that nobody wants; or that nobody can learn or use; or that don't meet enough user needs to make them pay money, which is after all what makes us professional developers.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Handling Image Files with TIFF

A picture may be worth more than 1,024 words but only if everyone can share the view.

by Anthony Meadow, Rocky Offner, and Michael Budiansky

It may sound categorical, but it's true: Anyone who is writing software that works with bit-mapped images should support the Tag Image File Format (TIFF). Why? Simply because TIFF has become the industry-standard format for storing bit-mapped images; the format is supported by almost all the desktop-publishing and scanner software in both the Macintosh and MS-DOS communities.

TIFF came into existence through the cooperation of several companies, especially Microsoft and Aldus. Many other companies have joined in the effort to develop and support this file format, including most of the scanner manufacturers (such as DEST Corp., Datacopy, and Hewlett-Packard) and the developers of desktop publishing software (Letraset, PS Publishing, and so on). The most recent version is Revision 4, released on April 31, 1987. Revision 5 is now under discussion and adds several important features.

The primary reason for the success of TIFF is the rapid rise of desktop publishing (DTP). DTP software and scanners became so popular that a way was needed to move images from scanners to publishing software. Initially, every scanner manufacturer developed a proprietary format for storing images, but it quickly became obvious that there was a better way to solve this problem. With the support of Aldus and Microsoft, TIFF became the alternative to supporting an odd collection of proprietary file formats. Fortunately, almost all scanner manufacturers decided to support TIFF, which has encouraged almost all developers of desktop-publishing software and paint programs to adopt TIFF, too.

Later in this article, we will describe the TIFF Library Package, which is now in the public domain. The package includes routines for reading and writing TIFF files, a sample application that uses them, and a utility program for examining TIFF files. Also included is a set of TIFF files that most TIFF implementations should be able to read. This set of files was designed to serve as an initial validation suite for TIFF implementations. You can write or call for the full package—the details are at the end of this article.

Anthony Meadow, Rocky Offner, and Michael Budiansky form the developer nucleus of Bear River Associates Inc., P.O. Box 1900, Berkeley, CA 94701.

What Can TIFF Do?

TIFF has satisfied many developers because it is capable of storing all the details about an image, which is not surprising because TIFF was developed as a superset of several existing proprietary formats.

TIFF is unlike most other file formats in that most information is not stored in fixed locations in the file. There are only 8 bytes of information in a TIFF file that have a specified location—the first 8 bytes in the file. Everything else is reached by using offsets from the start of the file. The categories of information that are currently supported seem to be sufficient for almost all applications today. If these categories are not sufficient, then others can be easily added. In fact, even proprietary information can be stored in a TIFF file without violating the specification.

Bit maps of any kind can be stored in a TIFF file—bilevel, grayscale, and color are all supported in Revision 5 of the TIFF standard. Images of any resolution, size, number of samples per pixel, and so on can be stored in TIFF files. These images might come from any kind of device, including scanners, facsimile machines, video frame stores, and so on, or from any kind of software that

can create a bit-mapped image.

TIFF is not supposed to be the only graphics file format. It may not provide sufficient capabilities for all bit-mapped graphics applications, although it's difficult to imagine what the exceptions might be. It does not provide support for object-oriented graphics, whereby images are composed of ellipses, rectangles, splines, and so on (as in Adobe Illustrator, MacDraw, or AutoCAD). TIFF also does not support PostScript or any other page-description language. The PICT and PICT2 formats supported by the Macintosh provide much of the functionality of TIFF with respect to bit maps and much more functionality to support object-oriented graphics and PostScript, but these file formats are proprietary. Table 1, below, compares TIFF with these and other file formats.

The Structure

The TIFF file structure is defined in a specification that is available from Microsoft or with the TIFF Library Package. Revision 4 is the latest, but Revision 5 is now in the works and adds conformance levels, a compression method for color and gray-scale images, and support for a "palette" type of color.

The obvious place to start learning about the TIFF file structure is to read the specification. After that, try dumping several TIFF files with td, the dump utility provided as part of the TIFF Library Package. Look at a variety of files in order to see what you can do with this format. The TIFF Library Package includes about two dozen files that show a wide variety of legal TIFF files.

Abbreviation	Name	Machines on which the format is supported				Support for				
		Mac	MSDOS	Unix	Other	Bilevel Bit Maps	Gray-Scale Bit Maps	Color Bit Maps	O-O Graphics	PostScript
TIFF	Tag Image File Format	Y	Y	Y	Y	Y	Y	Y	N	N
EPS	Encapsulated PostScript	Y	Y	Y	Y	Y[1]	Y[1]	Y[1,2]	Y	Y
PICT	(Macintosh) PICT	Y	N	N	N	Y	N	N	Y	Y[4]
PICT2	(Macintosh) PICT2	Y[3]	N	N	N	Y	Y	Y	Y	Y[4]
RIFF	Raster Image File Format	Y	N	N	N	Y	Y	Y	N	N
MacPaint	(Macintosh) MacPaint	Y	N	N	N	Y[5]	N	N	N	N

O-O graphics means object-oriented graphics (such as MacDraw).

Notes:
1–Bit-maps are supported, but not encouraged. Because EPS files are usually ASCII and not binary in form, bit-maps appear as large runs of digits.
2–Support for color in PostScript will improve when more color printers are available.
3–The PICT2 format is supported only on the Macintosh II at this time.
4–Support for PostScript in PICT and PICT2 files is through the use of special comments, so few applications can do anything with the PostScript commands directly.
5–MacPaint files are always 72 dpi and the maximum document size is 8 × 10 inches.

Table 1: *Comparison of image file formats*

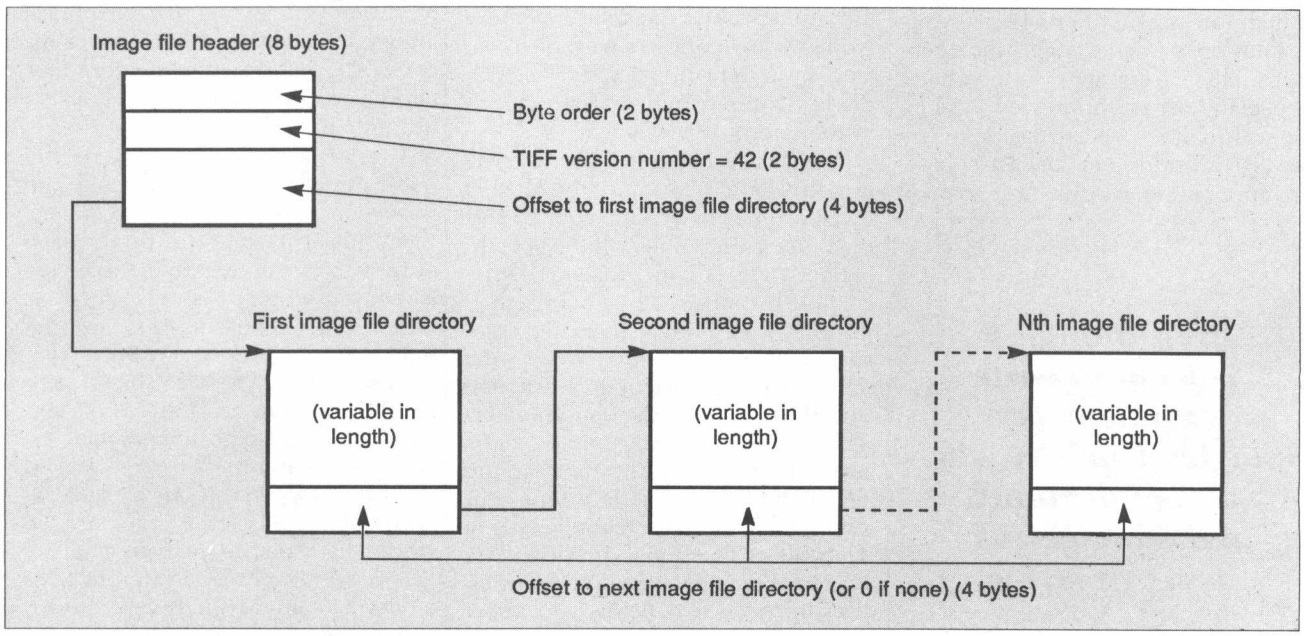


Figure 1: *Structure of a TIFF file*

TIFF

(continued from page 27)

A TIFF file is composed of three kinds of elements: an image file header, one or more image field directories, and collections of data. Figure 1, page 27, shows the general structure of a TIFF file. The image file header (IFH) always occupies the first 8 bytes in a TIFF file and is the only element that has a fixed position within the file. The image file header contains the byte order flag, the file version number (currently 42), and an offset to the first image file directory (IFD).

All elements other than the IFH have variable positions within the file and are located by using byte offsets from the start of file. A TIFF file is considered to be a continuous sequence of (8-bit) bytes numbered from 0. The largest a TIFF file can be is 232 bytes long, or about 4 Gbytes.

The TIFF specification, like most other specifications, is not especially easy to read. One misunderstanding that we've seen several times concerns the location of various components of the file. The first image file directory is not necessarily just after the image file header—it might be anywhere in the file. Its location is given in the image file header.

A TIFF file may include multiple versions of an image—for example, it might be useful to provide a lower-resolution screen version of a 300-dpi image (to present on a screen), especially if an optimized scaling algorithm was used to generate the lower-resolution version. Of course, an application would have to look for the lower-resolution version and display it rather than generate one on the fly from the 300-dpi version. As far as we know, there are no applications that write more than one version of an image to a file, perhaps because of the additional disk space required. At any rate, if there are multiple versions of an image, then each version is described by the information in its image file directory. Each IFD contains an offset to the next IFD or 0 if there isn't another one.

Even though it is legal to write an image as a single strip, virtually all applications write images as a set of strips

Directories and Tags

Before we look at what an IFD contains, let's look briefly at tags and tag entries. A tag entry is basically a chunk of data (or a field) with a name (or tag). A tag entry might point to the image, the image height, the image width, or the orientation of the image.

An image file directory contains all the tag entries for a version of an image, ordered by tag type. Figure 2, page 32, shows what an IFD looks like. Only one tag of each tag type is allowed in an IFD.

As shown in Figure 3, below, each tag is 12 bytes long, where the first 2 bytes are the tag type. There are almost 40 tag types, which are defined in the specification. You can use other tags, but you should contact the TIFF administrator at Microsoft first. It's important to reserve any nonspecified tags so that there won't be any conflicts with files created by other applications. The address of the administrator is given at the end of this article.

The next 2 bytes in the tag entry are the data type, and the next 4 bytes are the length (or count). Five data types are defined in the specification: *byte* (1-byte unsigned integer), *ASCII*, *short* (2-byte unsigned integer), *long* (4-byte unsigned integer), and *rational* (two *long*s—the first is the numerator of a fraction and the second is the denominator). The length field in the tag entry gives the length of the data for this tag in terms of the data type—for example, if the data has a length of 1 and is of type *long*, then the data is 4 bytes long. By using the information in these two fields, you know exactly how much data there is for this tag.

The last 4 bytes in the tag entry are either an offset to the data or, if the data occupies 4 bytes or less, the data itself. If the data is less than 4 bytes in size, then it is left-justified within the 4 bytes. This convention for the Offset field optimizes access to small chunks of information, although it does make the file structure more complex.

Stripped Data

Images in TIFF files are usually divided into strips to allow for the memory limitations of most machines. A strip is typically an integral number of scan lines (a scan line is usually a horizontal row). Most applications cannot fit an entire image in memory at one time—for example, an $8\frac{1}{2} \times 11$-inch bilevel image at 300×300 dpi requires more than 1 Mbyte of space. Most Macintoshes don't have this much space available, and most PCs don't have more than 640K of memory to start with.

Even though it is legal to write an image as a single strip, virtually all applications write images as a set of strips, where one strip typically is less than 64K in size. This allows applications to work with images using less memory than would otherwise be needed, although if enough memory is available, an application could still read a complete image into memory. When an image is divided into strips, the strips are either all compressed or all not compressed. The TIFF specification does allow you to compress some strips but not all.

The *StripOffsets* tag entry, as the name implies, contains offsets to the image data and not to the image. Each offset points to one image strip. This is true whether the image is composed of a single strip or many strips. These offsets are the only way to get to the image.

Gray-scale images are stored with all the bits for each pixel packed contiguously. Color images can be stored with all bits per pixel packed together or with the bits describing each color (in the RGB color model) stored in separate planes. The tag *SamplesPerPixel* has a value of 1 for a bilevel or gray-scale image, but for a color picture with three planes, it will have a value of 3. The *Planar-*

Configuration tag tells you whether there is one image plane or several—for example, if a color image was stored as multiple planes, there is one plane for red, a second for green, and a third for blue.

Two other tags always present in TIFF files are *ImageWidth* and *ImageLength*; they contain the width and height of the image in pixels. Each of these is usually 2 bytes long (although a program should always check the Data Type field in the tag entry to find out how long a field really is). For these two tags, the data is actually kept in the Offset field of the tag entry, rather than being pointed to by a file offset.

Compressing Images

Image files are large compared with the average word-processing or spreadsheet document. Now that gray-scale scanners are on the market (and color scanners can't be too far away either), image files could become much larger. Table 2, this page, gives the size of various images at different resolutions. In order to conserve users' disk space, applications should compress images.

Revision 4 of the TIFF standard specifies several compression schemes for bilevel images. Revision 5 adds a method for compression of color and gray-scale. (The compression methods defined in Revision 5 are listed in Table 3, below.) Each of these methods is lossless—that is, each preserves all information in the image. Compression methods are possible that produce much higher compression ratios, but they do not save all of the image's information. These methods could be supported in a TIFF file, but no one has seen the need for them yet. An application needs to support only one compression method when writing each type of image (bilevel, gray-scale, or color) but should be able to read an image in any of the other compression methods.

The default compression method in the specification isn't really compression; it's simply packing data into bytes as tightly as possible. One of the other methods is a variation on this, where the data is packed into (16-bit) words as tightly as possible.

The compression methods for bilevel images are derived from the CCITT (International Telegraph and Telephone Consultative Committee) standards developed for facsimile machines. These methods are based on a Huffman run-length code. The CCITT arrived at the code by looking at samples of typical documents sent by facsimile. It's quite likely that images used in desktop publishing are not like the documents used in developing the CCITT standards, so the compression methods are probably not optimal. They aren't too bad, however; a compression ratio of 4 to 1 is typical. The TIFF standard allows for additional compression schemes in the future, but the TIFF file I/O code will have to be revised in all applications so that they can read files that use these new compression methods.

Why Use TIFF?

The most obvious reason to use TIFF is that everyone (well, almost everyone) already does. If you are writing an application that works with bit-mapped images, it can work with all the existing applications that already produce or read TIFF files. If you develop a proprietary file

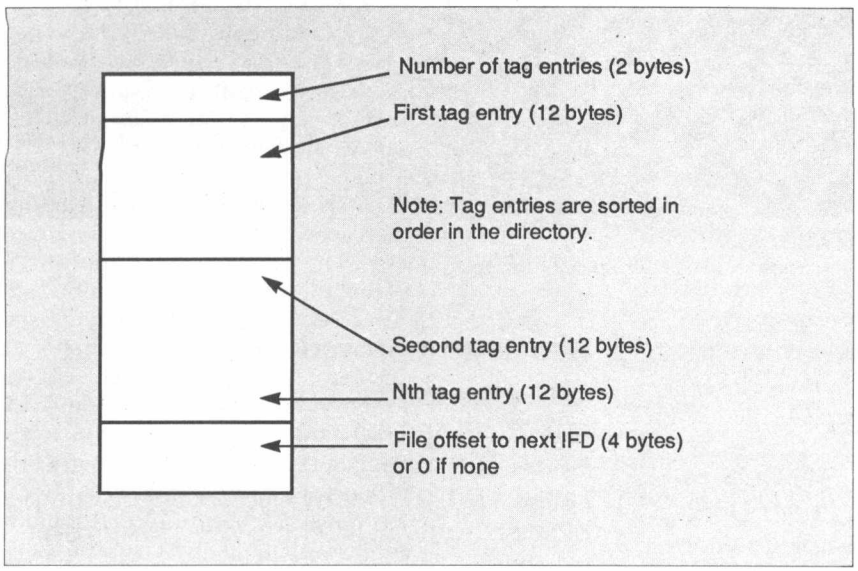


Figure 2: *Structure of an image file directory*

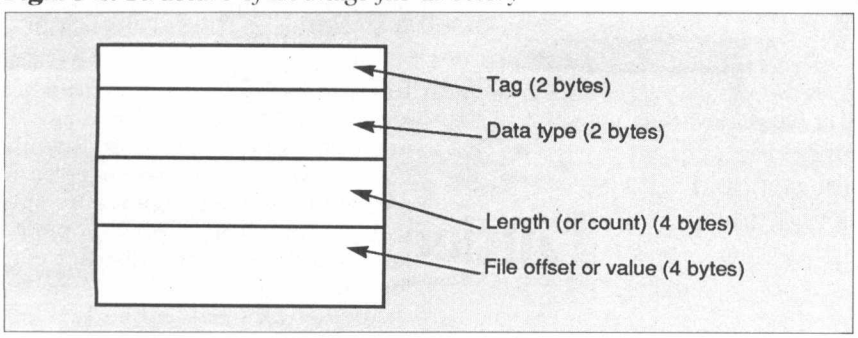


Figure 3: *Structure of a tag entry*

The richness of the file structure has caused a couple of other problems.

(continued from page 33)

format, you will have to talk many others into supporting it.

Another advantage of TIFF is that it was designed to support changes easily. By using new tags, additional *information* can be added to TIFF files. Older applications won't be able to take advantage of the information in the new tags, but as long as the information they need is there, they can still use the file. Hopefully you won't require any new tags because TIFF already provides a rich set. The kind of information that TIFF supports is a superset of virtually all that contained in proprietary image file formats.

It's also possible to store proprietary information in a TIFF file. The tags numbered 32,768 to 65,535 are reserved for this purpose. Developers who would like one or more tags reserved should contact the TIFF administrator at Microsoft. Obviously, only applications "in the know" are able to use such proprietary information.

Resolution* (dpi)	Bits/Pixel				
	1	4	8	12	24
72	61	242	485	727	1,454
150	263	1,052	2,104	3,156	6,311
300	1,052	4,208	8,415	12,623	25,245
600	4,208	16,830	33,660	50,490	100,980
1,200	16,830	67,320	134,640	201,960	403,920

* Assumes that all images are 8.5 × 11 inches in size
** Assumes that the horizontal and vertical resolution are the same

Table 2: *Sizes of uncompressed bit-mapped images** (K)*

Tag Value	Name
1	None (but pack data into bytes tightly)
2	CCITT Group 3 1-Dimensional modified Huffman run length encoding
3	Facsimile-compatible CCITT Group 3
4	Facsimile-compatible CCITT Group 4
5	Differential run-length encoding
32771	None (same as 1 except pack tightly into words)
32773	PackBits compression

Table 3: *Compression methods in TIFF, Revision 5*

Portability of data is becoming more and more important these days. It's now common to see Macs and PCs connected over a local-area network. TIFF supports portability of data because both the Motorola and Intel differences are clearly defined and can therefore be handled easily by an application. TIFF files can also be easily moved to almost any other file system because TIFF makes no assumptions about the underlying file system.

Problems with TIFF

TIFF does have a few problems, but as the standard evolves, many of them are being solved.

The TIFF file structure is not simple—it is more complex than many existing proprietary file formats, such as MacPaint's. This complexity costs time in several ways—for example, there is more overhead to write a TIFF file than a file with a simpler format. It also takes longer to write the TIFF I/O functions for an application because of TIFF's generality, although the library described in this article will reduce this development time for Macintosh developers. There are also TIFF toolkits available from others for MS-DOS machines.

The richness of the file structure has caused a couple of other problems. The TIFF standard (*until Revision 5*) did not specify a minimal set of tags, so each developer has used a different subset of tags in his or her files. This problem should be solved in Revision 5 of the standard, which specifies six conformance levels. Each level is specified by listing which tags and compression methods must be supported. Hopefully all developers will make the (few) changes in future revisions of their products to bring them to a reasonable conformance level.

Table 4, page 39, outlines the features of the various conformance levels. The FAX conformance level is in a different category from the others because the features required to support facsimile machines are special. An application that supports level 3 and FAX is described as conforming to level 3+FAX.

Another problem is that there are no compression schemes for color and gray-scale images. An 8.5 × 11-

inch gray-scale image with 8 bits per pixel at 300-dpi resolution requires 8.4 Mbytes of storage without compression. Even people with hard disks would quickly run out of room without some form of compression for these images. This problem is also being solved in Revision 5 of the TIFF standard, which details a compression method for both color and gray-scale.

Tools from the Library

Necessity is the mother of invention. The TIFF Library Package was created as part of DEST Corp.'s product Publish Pac, which is a Macintosh application that lets users operate one of DEST's scanners to read in images and text. (There is also a version of Publish Pac for the IBM PC.) DEST was an early adopter of TIFF; it decided to use TIFF as its standard file format rather than develop yet another proprietary format. The TIFF Library Package is used by Publish Pac to read and write TIFF files. In its current state, it is used in the latest version of Publish Pac. It is written in MPW C (from Apple's Macintosh Programmer's Workshop) and could be ported to other versions of C on the Macintosh with little trouble.

The TIFF Library Package provides low-level routines for working with TIFF files. These routines provide a standard way to read and write TIFF files as well as to manage TIFF tags. The TIFF file format requires a certain amount of bookkeeping, such as ordering all tags sequentially. The library routines handle this automatically to ensure that all images are consistent with the specification. If you use this library, you won't have to learn all the low-level details of what TIFF files look like, but you will still have to decide which tags you want to read and write. The library includes routines to read and write the TIFF header, read and write the tags as a group, and read and write images. Table 5, page 43, contains a list of all the function names.

Reading and writing the file header is straightforward. The two routines *TReadHeader* and *TWriteHeader* read and write the TIFF file header, which contains the byte ordering of the file (Motorola or Intel) and the offset of the first image file directory. If the byte ordering is different from the native ordering, the offset to the directory and all subsequent numerical values are adjusted before they are returned, so an application doesn't have to know whether the file originated on a Macintosh or a PC.

To facilitate tag handling, an in-memory, tag-management scheme was designed so that all tags and their values are read from the file into memory. From there the tags and their values can easily be located, modified, and removed, and new tags can be inserted with simple function calls. When reading a TIFF file, call *TReadTags* to get all the tags into memory at once. You can then make function calls to check for the presence of a tag or get a tag's value.

When you're ready to create a TIFF file, make calls to create or modify the appropriate tags. The position of the tags and the method for storing the values according to the TIFF specification is handled by the library routines. A call to *TPutPtrTag* or *TPutHdlTag* is made for each tag that's being added to the TIFF file, with two exceptions. The *StripOffsets* and *StripBytesCounts* tags are created and updated automatically by the routine that writes the image strips to the file. Calls to *TWriteTags* include a pointer to the tag's value, and the tag is included in the in-memory tag list.

To write an image to a file, an application makes one call to *TWriteImageStrip* for each strip that will be in the TIFF file. Each strip must have a number of rows less than or equal to that specified by the *RowsPerStrip* tag. Reading an image from a file is a bit more flexible. Any number of rows can be requested by using the *TReadImage* function starting with any row, regardless of the number of rows per strip.

Two auxiliary routines, *TFixOddRowBytes* and *TUnfixOddRowBytes*, provide image adjustment. Using compression method 1, the bits of each row of an image are stored in the smallest possible number of bytes. Some systems normally fit the bits into the smallest number of (16-bit) words—for example, in the Macintosh operating system, bit

Level	Name	Byte Order Supported	No. of Images (IFDs) per File	No. of Tags That Must Be Supported	Bilevel Compression Methods	Gray-Scale Compression Methods	Color Compression Methods
1	Bilevel images with simple compression	Native	1	6	1, 32773	N/A	N/A
2	Rich bilevel images with CCITT compression	Both	1	10	1, 2, 32771, 32773	N/A	N/A
3	Compressed gray-scale images	Both	1	16	1, 2, 32771, 32773	1, 5, 32771	N/A
4	Compressed color images	Both	1	18	1, 2, 32771, 32773	1, 5, 32771	1, 5, 32771
FAX	Bilevel facsimile data	Both	Many	13	1, 2, 3, 4, 32771, 32773	N/A	N/A
5	Unlimited support	Both	Many	37	All defined	All defined	All defined

Table 4: *Conformance levels in TIFF, Revision 5*

maps are always an even number of bytes in length. These auxiliary routines are provided to translate the image between these two storage methods if the bits in a row of an image can fit in an odd number of bytes.

The current TIFF Library Package supports most of the first and second conformance levels. Only compression methods 1 (tight packing into bytes) and 2 (modified Huffman run-length encoding) are supported at this time. Also, only the default orientation is supported, where the 0th row represents the top of the image and the 0th column represents the left side of the image (there are seven other possible orientations). When Revision 5 of the TIFF specification is released, the package will probably be upgraded to meet conformance level 3.

The Sample Program

The sample program demonstrates how to use the routines in the Macintosh version of the TIFF Library Package. The program is limited in its ability to display images or read in complex nonbilevel TIFF images, but it does use most of the functions in the library. It can read and write TIFF files and will let you cut and paste Macintosh PICT images to and from the Clipboard—for example, a drawing can be made in MacPaint, cut or copied into the Clipboard, and then pasted into the sample program and written out to a TIFF file. Similarly, a TIFF image can be read into the program, copied into the Clipboard, and pasted into other programs such as MacPaint.

Listing One, page 54, shows two functions—*ReadTiff* and *WriteTiff*—from the sample program. We will now use them to demonstrate the use of the library functions.

The ReadTiff Function

The *ReadTiff* function shows how to use the TIFF library routines to read a TIFF file. First, you read the file header using *TReadHeader*, and then you read all the tags into memory using the *TReadTags* function.

Subsequently, local data structures are filled with the values of the tags via calls to the tag-management routines *TFindTag* and *TGetTag*. If a tag is present, then the value is set; if not and the tag has a default value, the local data for that item is set to the default. If there is no default for a tag, then you either ignore that value or report an error if the tag is necessary.

Once the tag values have been obtained, you determine if there are any values that require facilities beyond those provided to display this image. If so, an error is reported. Once you've determined that the image as described by the tags is correct and that you are able to display that image, a call to *TReadImage* is made. The sample program requests the lines of the image from line 0 through the number of rows in the entire image or the number of rows that will fit into 32K of memory, whichever is smaller. If you successfully read the image, then it (or some portion of it) is displayed in a simple window.

The WriteTiff function

The *WriteTiff* function demonstrates how to write a TIFF file using functions from the TIFF library. A subset of the available tags is placed in the tag list using the *TPutPtrTag* function. The tags specified are those required by the specification plus a couple of others that, although not required by the TIFF specification, are quite useful.

After placing all the tags you want in the tag list, the strips are written

File-handling functions
- TReadHeader
- TWriteHeader
- TReadTags
- TWriteTags
- TReadImage
- TWriteImageStrip

Tag list management functions
- TFindTag
- TGetTag
- TPutPtrTag
- TPutHdlTag

Auxiliary functions
- TFixOddRowBytes
- TUnfixOddRowBytes

Table 5: *TIFF Library Package functions*

(continued from page 43)

to the file using the *TWriteImageStrip* function. The number of rows per strip (8) used by the sample program is an arbitrary value chosen to demonstrate how to write multiple strips. The *RowsPerStrip* tag can be set to any value the creator desires, although it is a good idea not to let strips get too big. For the Macintosh, it is wise to limit strips to no more than 32K.

Once all the strips have been written, the tags are flushed to the file by a call to *TWriteTags*. Finally, the header is written using *TWriteHeader*. After the file has been closed and flushed to disk, you've created a TIFF file.

Conclusion

The TIFF file format has become popular in both the Macintosh and MS-DOS worlds. It offers a way to share any kind of bit-mapped image with a large and growing number of other applications. Anyone who is writing software that works with bit maps should support TIFF. The TIFF Library Package described in this article will help any Macintosh programmer read and write TIFF files with a minimum of work. It can also be ported to other machines.

Availability

Because of the size of the TIFF Library Package (close to 1 Mbyte), we've been forced to forego the usual distribution method of printing the listings in the magazine or putting them on CompuServe. Serious developers and readers of *DDJ* can, however, receive a free set of disks for the TIFF Library Package by mentioning this article in a request to one of two sources.

For a copy of the TIFF Library Package for the Macintosh, send your name and address to Bear River Associates Inc., Attn: TIFF, P.O. Box 1900, Berkeley, CA 94701 or call 415-644-9400 (9 A.M. to 5 P.M. PST) and ask for the TIFF Library Package.

A similar TIFF package is available for the PC. To get this package, write to DEST Corp., Attn.: Debra Levesque, 1201 Cadillac Ct., Milpitas, CA 95035 or call 408-946-7100 and ask for Debra Levesque.

To receive a copy of the latest version of the TIFF standard, to reserve a proprietary tag, or to comment on the standard, write to Manny Vellon, Windows Marketing Group, Microsoft Corp., 16011 N.E. 36th Wy., P.O. Box 97017, Redmond, WA 98073-9717.

Bibliography

Andrews, Nancy; and Fry, Stan. "TIFF: An Emerging Standard for Exchanging Digitized Graphics Images." *Microsoft Systems Journal* (July 1987): 71–76.

DDJ

(Listing begins on page 54.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Virtual Arrays in C

The concept of virtual arrays is not new, but it can now be applied to microcomputers.

by Mark Tichenor

This article presents a virtual-array management scheme for C programmers. This simple and flexible approach uses the power of the C language to manipulate arrays that extend themselves automatically and that are limited in size only by the file-size constraints of the operating system. The concept of virtual arrays is not new, but it can now be applied to microcomputers.

Highlights of the scheme are:

- Virtual arrays are stored on disk but are accessed as though they are in memory.
- Automatic file management is provided without explicit reads or writes.
- Disk records can be accessed simply by referencing an array element in any C expression.
- Data can be written to any record in the file simply by using an assignment statement to place a value in an array element.

The Problem

Many problems lend themselves naturally to simple solutions based on the use of data arrays because arrays are easy to manipulate using simple notation. Data arrays can only be as big as available memory allows, however. This is an aggravating limitation because, once arrays outgrow available memory, they are no longer useful.

In implementing an application prototype involving complex tree structures, I chose an array approach using data arrays that incorporated pointers. These pointers were the array indices of other array elements. As the application provided the means to manage a growing volume of information, it was necessary to overcome the memory limitation imposed by the array model.

A lot of thought went into solutions based on dynamic memory allocation of data structures as this approach appeared to promise an elegant and general solution. The need to swap data between memory and disk still remained, however. In fact a whole new problem arose: that of resolving the differences between pointers in memory and pointers on disk.

When dynamically allocated data structures are linked by memory pointers, these pointers become meaningless when written to disk. The linkages must be recreated when the structures are brought back into memory. This problem dramatically increases the complexity of any program that incorporates the swapping of data between memory and disk.

It would be a great advantage to keep using data arrays if their size were not limited by available memory.

The Solution

The memory limitation associated with arrays can be overcome by using virtual arrays that are stored on disk and accessed as though they were in memory. Thus, virtual arrays offer a simple, elegant solution. The C programming language provides the power to manage virtual arrays automatically through the creative use of pointer notation and *#define* macros.

It turns out that virtual-array elements can be accessed by simple reference using a predefined alias and an array index—for example, *item_number(i) = 5;* could be used to set the *item_number* field of the *i*th array element to the value *5*.

Because virtual arrays reside on disk, with paging to memory automatically accomplished behind the scenes, you have a disk-based data management system that operates without any explicit reads or writes. These operations are performed by the virtual-array access function, which is invoked by the predefined access macros.

Mark Tichenor is an instructor in the Business Information Processing Dept., Holland College, Charlottetown Centre, Weymouth St., Charlottetown, PE C1A 4Z1, Canada. He is currently developing interactive software design tools.

The Code

Listing One, page 63, contains the virtual-array header file, and Listing Two, page 63, contains the virtual-array access routines. Four C functions handle all the mechanics of managing the virtual arrays.

The Initialization Routine

```
int init_v_array(filename,
                 record_size, filchar)
char *filename, filchar;
int record_size;
```

Init_v_array creates a new virtual-array file named by filename. File headers are written initializing the record size of the file to *record_size* and setting the number of records to 0. The specified fill character is also placed in the file header for later use in initializing new array elements. The file is then closed. This routine returns a value of 1 if successful, 0 if not.

An example is:

```
if (init_v_array("DATA.VAR",128))
                    printf("Success!");
```

which will try to create a new virtual array named *"DATA.VAR"* with elements 128 bytes long and will print a message if successful.

The Open Routine

```
VACB *open_v_array(filename,
                   buffer_size)
char *filename;
int buffer_size;
```

open_v_array prepares an existing virtual array for use. *buffer_size* specifies how many array elements to allocate space for in memory. The routine returns *NULL* if unsuccessful; otherwise, it returns a pointer to the created virtual-array control block (*VACB*).

An example is:

```
VACB *item array;
item_array =
   open_v_array("DATA.VAR",100);
```

which opens the *"DATA.VAR"* array file and reserves enough buffer space for 100 array elements.

The Close Routine

```
void close_v_array(array)
VACB *array;
```

close_v_array writes elements from buffer to disk, closes the array file, and frees allocated memory.

An example is:

```
close_v_array(item_array);
```

The Access Routine

```
void *access_v_array(array,index)
VACB *array;
long index;
```

This routine performs the low-level file management for virtual arrays. It makes sure the array element referenced by *index* is in memory and returns a pointer to it in memory. If the specified array element is already in memory, access is immediate; if not, it is read into memory after saving the record it displaces in the buffer. If the referenced array element does not exist, the routine automatically extends the *array* file so that it does.

For an example, see the listings and the section entitled "Access Notation."

The Access Algorithm

The key to easy reference to array elements (or to fields within them) is the *access_v_rec* function. This function returns a *void* pointer to the location of the element in memory after making sure it is there.

In the interests of demonstrating the feasibility of this approach quickly, I have paid no regard to optimization. The only stipulation was that access to array elements already in memory be as fast as possible.

To meet this requirement, I chose a simple modulus buffering scheme. Each array element has a fixed position in the buffer calculated by element number modulus buffer size. Each element in the buffer is prefixed with a *long* array index containing the number of the element

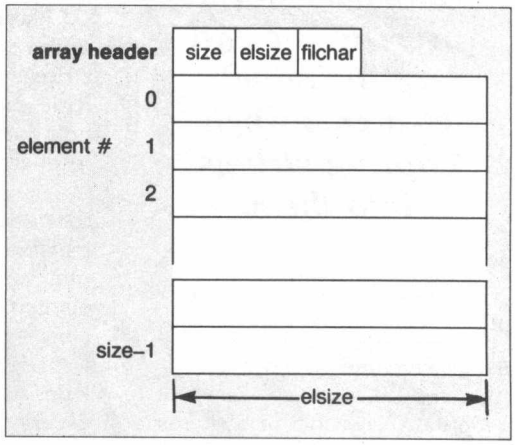


Figure 1: *Virtual-array file layout*

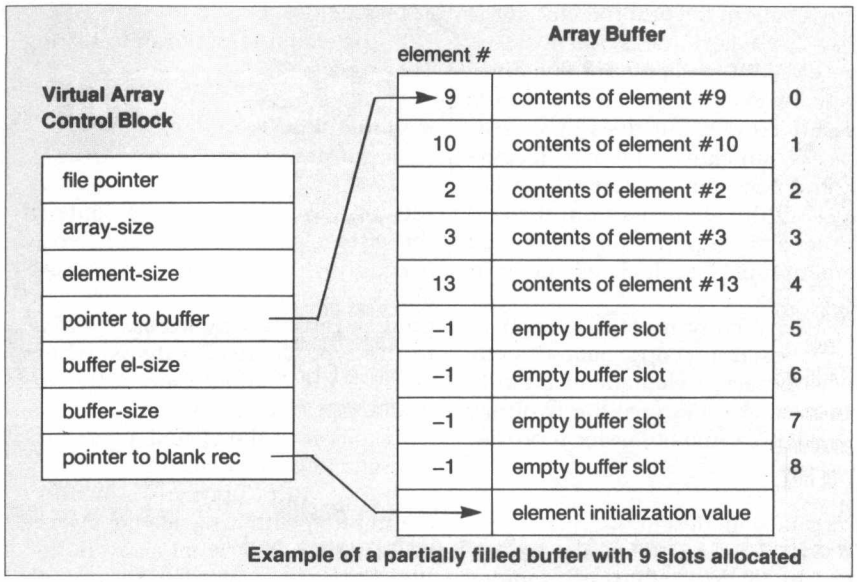


Figure 2: *Virtual-array control block and array buffer*

(continued from page 47)

currently present or a –1 if no element is in that buffer position.

If the referenced array index does not match the index of the element currently occupying the calculated buffer location, that array element in the buffer (if any) is written to disk and the referenced element is read in before the buffer address is returned. This scheme avoids searching for records in memory.

Cautions

You should use only *long* integer indices to access array elements. Be sure to *#include* <varray.h>.

Take care not to overrun the end of the array elements when copying strings into them.

You should also take care not to accidentally reference array items far beyond the end of the array unless you really intend to do so. When an array element beyond the end of the array (that is, it does not exist) is referenced, the access routine automatically extends the file so that the referenced element does exist. This process can cause a lot of disk activity and take considerable time because all the elements from the end of the array up to the referenced element must be initialized and written to disk.

Memory copy functions, such as *strcpy*, used to copy data directly from one array element to another are unreliable because the two may occupy the same buffer location. For example:

```
strcpy(desc(n),desc(m));
```

will not work when *n modulus buffer_size* equals *m modulus buffer_size*. However:

```
qty(n) = qty(m) + 1;
```

will work because memory copying is not invoked. (This is true because the compiler calculates the assignment value before it calculates the address for assignment.)

To avoid buffer collision, use:

```
strcpy(temp_desc,desc(m));
strcpy(desc(n),temp_desc);
```

The buffer collision problem associated with the simple modulus buffering scheme could be avoided with the development of a robust least recently used buffering scheme.

Buffered access of any kind also precludes the use of such things as in-memory sort utilities (for example, *qsort*).

Take care not to overrun the end of the array elements when copying strings into them. This would corrupt the index information in the next buffer slot, with unpredictable (probably bad) results.

Access Notation

#define statements are used to simplify access notation both to array elements and to fields within them. One #define is required for each virtual array used. Optionally, one #define may be used to simplify access to each field within a virtual array of structures. Alternately, pointer notation may be used to access fields.

In the example program in Listing Three, page 66:

```
#define VREC(i) ((items *)
        access_v_rec(item_array,i))
```

sets up easy access to elements in the virtual array called item_array (stored in *"ITEMS.VAR"*). In this example, *(items *)* casts the *void* pointer returned by *access_v_rec* to the type *pointer to items*, where *items* is the defined type of the virtual-array element structure. *item_array* is the virtual-array handle returned by *open_v_array*.

In use, *VREC(i)* returns a memory pointer to the *i*th array element wherever it is in the virtual-array buffer. **VREC(i)* is a reference to the *i*th *items* structure in the array. It can

be used in any expression as a variable of type *items*.

The expression:

```
#define item(i) VREC(i)->v_item
```

in Listing Three sets up easy access to the *v_item* field in the *i*th element of the *item_array* virtual array. *item(i)* becomes its alias. This allows you to say:

```
item(i) = 24;
```

which is much more intuitive than:

```
((items *)
access_v_rec(item_array,i))->
                        v_item = 24;
```

This ease of reference is what makes this virtual-array system possible.

The size of a virtual array is always available because it is stored in the virtual-array control block. The expression:

```
i = item_array->size;
```

sets *i* to the current size of the virtual array referred to by *item_array*. You should never change this value but can refer to it to determine the index value to use in order to extend the array by one element. For example:

```
i = item_array->size;
item(i) = any_value;
```

will extend the array by one element.

Multidimension arrays are also possible; however, you can extend them only in one dimension. In the example program, the *v_desc* field is a character array that can be regarded as two dimensional when extended over the virtual array. Particular elements in the two-dimensional array can be referenced by *desc(y)[x]*, where *y* is the element number and *x* is the character number.

If you wished to formalize this two-dimensional access, you could add a new macro:

```
#define D(x,y) VREC(y)->v_desc[x]
```

in which *D(x,y)* would refer to column *x*, row *y*. This construct is useful for creating very large virtual-display spaces that can be quickly mapped to the physical display to accomplish fast panning.

Using Virtual Arrays

1. Create a virtual-array file by calling *init_v_array* and passing it the DOS file name, the record length of array elements (in bytes), and a fill character for initializing new elements in the array as they are referenced. This creates a new file, overwriting any existing file of the same name, and writes out the array header information: 4 bytes for array size (initialized to 0), 2 bytes for the size of array elements, and 1 byte for the fill character. Perform this step only once.
2. Typedef your array element structure.
3. Define your array element access macro—for example:

```
#define VREC(i) ((items *)
        access_v_rec(item_array,i))
```

4. Define each field's access macro for the array.
5. Call *open_v_array*, passing it the file name and buffer size. This call must assign the return value to a variable of type *(VACB *)*. This must be the same variable name used in your array access macro—for example:

```
item_array =
    open_v_array("ITEMS.VAR",10);
```

6. Do whatever you want with your virtual array using the access macros you defined in steps 3 and 4.
7. Close your virtual array by calling *close_v_array* and passing it your virtual-array handle—for example, *close_virtual_array(item_array);*.

Implementation

These routines were tested using Borland's Turbo C development system with both the large- and small-memory models. Using a standard 4.77-MHz PC-compatible with a cheap (slow) hard disk and MS-DOS 3.2, 200 records of 128 bytes each were swapped in 15 seconds and 200 4-byte records were swapped in less than 2 seconds. Though these access speeds may be slow for massive matrix multiplication, they are adequate for many data-access applications.

The beauty of this virtual-array system is that arrays are no longer limited by available memory and all the mechanics of file management take place automatically behind the scenes.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 63.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Listing One *(Text begins on page 26.)*

```
/* Primary Interface Files */
#include "Types.h"
#include "Quickdraw.h"
#include "Windows.h"

/* Other Interface files */
#include "Errors.h"
#include "Files.h"
#include "Memory.h"
#include "Packages.h"
#include "Scrap.h"

/* Application-specific Include files */
#include "::TiffLibrary:TIFFLib.h"
#include "sample.h"
#include "messages.h"

static Ptr  SetIDPtr();
static void CleanUp();
static void InitID();

#define     MAXIMAGESIZE    0x8000                          /* Limit images to 32K for now */
#define     INFINITY                        0x4000000       /* TIFF Spec says 2**32-1 but this is big enough */

/* Read in an image from a TIFF format file.  As the code demonstrates, we
 * do not read in very complicated images.  We read in a number of tags,
 * and reject the image as an unsuitable tiff file if any of several
 * conditions exist.  We do not read in images that have more than one bit
 * of image data per pixel.  Of those simple images that we do read, we
 * will only read the first 32k of that image.
 */
Boolean ReadTiff(refNum, myBitMapPtr)
Int16 refNum;
BitMap *myBitMapPtr;
{
            Handle                                          listH;
            Boolean                                         oddRowBytes;
            Int8                                            dummy;
            Int16                                           byteOrder,
                                                                        rowBytes,
                                                                        scrnHRes,
                                                                        scrnVRes;
            Int32                                           tagOffset,
                                                                        nextDirOffset,
                                                                        nextFileFree,   /* next free location in output file */
                                                                        dirOffset,
                                                                        rowsPerImage,
                                                                        count,
                                                                        size;
            Rational                        xRes,
                                                                        yRes;
            Rect                                            imageRect;
            TiffDirEntry    tagDirEntry;
            id                                                          id;                                             /* image
description */

            InitID(&id);
            ScreenRes(&scrnHRes, &scrnVRes);                /* if needed for defaults */

            /*
             * Read in header and Tags
             */

            if (TReadHeader(refNum, &dirOffset, &byteOrder) != noErr) {
                        ErrorMessage(BADREADHEADER);
                        return(false);
            }

            if(TReadTags(refNum, byteOrder,
                                                            &listH, dirOffset, &nextDirOffset) != noErr) {
                        ErrorMessage(BADREADTAGS);
                        return(false);
            }

            /* Get tags values.
             */
            /* SUBFILE_TYPE_TAG */
            if (TFindTag(listH, &tagOffset, SUBFILE_TYPE_TAG))
                        TGetTag(listH, tagOffset, &id.subfileType, sizeof(id.subfileType));
            else {
                        ErrorMessage(BADTIFF);
                        CleanUp(listH, &id);
                        return(false);
            }
            /* IMAGE_WIDTH_TAG */
            if (TFindTag(listH, &tagOffset, IMAGE_WIDTH_TAG))
                        TGetTag(listH, tagOffset, &id.imageWidth, sizeof(id.imageWidth));
            else {
                        ErrorMessage(BADTIFF);
                        CleanUp(listH, &id);
                        return(false);
```

(continued on page 56)

Listing One *(Listing continued, text begins on page 26.)*

```
        }
        /* IMAGE_LENGTH_TAG */
        if (TFindTag(listH, &tagOffset, IMAGE_LENGTH_TAG))
                TGetTag(listH, tagOffset, &id.imageLength, sizeof(id.imageLength));
        else {
                ErrorMessage(BADTIFF);
                CleanUp(listH, &id);
                return(false);
        }
        /* ROWS_PER_STRIP_TAG */
        if (TFindTag(listH, &tagOffset, ROWS_PER_STRIP_TAG)) {
                TGetTag(listH, tagOffset, &id.rowsPerStrip, sizeof(id.rowsPerStrip));
                tagDirEntry = GetDirEntry(listH, tagOffset);
                switch(tagDirEntry.type) {
                case LONG:
                        break;
                case SHORT:     /* ok, but convert returned value */
                        id.rowsPerStrip = (long)( *((Int16 *)(&id.rowsPerStrip)) );
                        break;
                default:
                        ErrorMessage(BADTIFF);
                        CleanUp(listH, &id);
                        return(false);
                }
        }
        else {
                id.rowsPerStrip = INFINITY;
        }
        /* SAMPLES_PER_PIXEL_TAG */
        if (TFindTag(listH, &tagOffset, SAMPLES_PER_PIXEL_TAG))
                TGetTag(listH, tagOffset, &id.samplesPerPixel, sizeof(id.samplesPerPixel));
        else {
                id.samplesPerPixel = 1;
        }
        /* BITS_PER_SAMPLE_TAG */
        id.bitsPerSample = SetIDPtr(listH, BITS_PER_SAMPLE_TAG, 1L, SHORT);
        if (id.bitsPerSample == nil) {
                        CleanUp(listH, &id);
                        return(false);
        }
        /* PLANAR_CONFIG_TAG */
        if (TFindTag(listH, &tagOffset, PLANAR_CONFIG_TAG))
                TGetTag(listH, tagOffset, &id.planarConfig, sizeof(id.planarConfig));
        else
                id.planarConfig = 1;
        /* COMPRESSION_TAG */
        id.compression = SetIDPtr(listH, COMPRESSION_TAG, 1L, SHORT);
        if (id.compression == nil) {
                        CleanUp(listH, &id);
                        return(false);
        }
        /* MIN_SAMPLE_VALUE_TAG */
        id.minSampleValue = SetIDPtr(listH, MIN_SAMPLE_VALUE_TAG, 0L, SHORT);
        if (id.minSampleValue == nil) {
                        CleanUp(listH, &id);
                        return(false);
        }
        /* MAX_SAMPLE_VALUE_TAG */
        id.maxSampleValue = SetIDPtr(listH, MAX_SAMPLE_VALUE_TAG,
                                                                        (Int32)((1 <<
*id.bitsPerSample) - 1), SHORT);
        if (id.maxSampleValue == nil) {
                        CleanUp(listH, &id);
                        return(false);
        }
        /* PHOTOMETRIC_INTERP_TAG */
        if (TFindTag(listH, &tagOffset, PHOTOMETRIC_INTERP_TAG))
                TGetTag(listH, tagOffset, &id.photoInterp, sizeof(id.photoInterp));
        else {
                id.photoInterp = 0;             /* assume mac photometric interpretation */
        }
        /* FILL_ORDER_TAG */
        if (TFindTag(listH, &tagOffset, FILL_ORDER_TAG))
                TGetTag(listH, tagOffset, &id.fillOrder, sizeof(id.fillOrder));
        else
                id.fillOrder = 1;
        /* ORIENTATION_TAG */
        if (TFindTag(listH, &tagOffset, ORIENTATION_TAG))
                TGetTag(listH, tagOffset, &id.orientation, sizeof(id.orientation));
        else
                id.orientation = 1;
        /* X_RESOLUTION_TAG */
        if (TFindTag(listH, &tagOffset, X_RESOLUTION_TAG))
                TGetTag(listH, tagOffset, &id.xResolution, sizeof(id.xResolution));
        else {
                id.xResolution.numerator = (Int32)scrnHRes;
                id.xResolution.denominator = 1;
        }
        /* Y_RESOLUTION_TAG */
        if (TFindTag(listH, &tagOffset, Y_RESOLUTION_TAG))
                TGetTag(listH, tagOffset, &id.yResolution, sizeof(id.yResolution));
```

(continued on page 59)

Listing One *(Listing continued, text begins on page 26.)*

```
            else {
                        id.yResolution.numerator = (Int32)scrnVRes;
                        id.yResolution.denominator = 1;
            }

            /* Initialize of non-tag values.
             */
            oddRowBytes = (((id.imageWidth * (*id.bitsPerSample)) + 7) / 8) % 2 != 0;
            id.stripsPerImage =
                                                                    (id.imageLength + id.rowsPerStrip - 1) / id.rowsPerStrip;

            /*              Check Tag Values to see if we can read this TIFF file.
             *
             *              NOTE: Although the majority of the tag were read, all the values
             *              obtained are not used in displaying the image in THIS PROGRAM.
             *              Those not used were read in simply to provide the example.
             */
            if ( (id.samplesPerPixel != 1)                                                                          ||
                         (*id.bitsPerSample != 1)                                                                    ||
                         (id.planarConfig != 1 && id.planarConfig != 2) )              {
                        ErrorMessage(BADTIFF);
                        CleanUp(listH, &id);
                        return(false);
            }
            if (id.photoInterp != 0)        /* We don't translate yet so let 'em know */
                        ErrorMessage(BADPHOTOINTERP);

            rowBytes = (((id.imageWidth - 1) / (2 * 8)) + 1) * 2;
            /* only make image as much as will fit in MAXIMAGESIZE for now */
            size = id.imageLength * rowBytes;
            if (size > MAXIMAGESIZE) {
                        ErrorMessage(IMAGECROPWARN);
                        size = MAXIMAGESIZE;
            }
            rowsPerImage = size / rowBytes;

            /* Prepare bitmap.
             */
            if (myBitMapPtr->baseAddr != nil)
                        DisposPtr(myBitMapPtr->baseAddr);
            if ((myBitMapPtr->baseAddr = MyNewPtr(size)) == nil) {
                        CleanUp(listH, &id);
                        return(false);
            }
            myBitMapPtr->rowBytes = rowBytes;
            myBitMapPtr->bounds.top = 0;
            myBitMapPtr->bounds.left = 0;
            myBitMapPtr->bounds.bottom = rowsPerImage;
            myBitMapPtr->bounds.right = id.imageWidth;


            /* Read in image.
             */
            if (TReadImage(refNum, listH,
                                                                    0L, myBitMapPtr->baseAddr, rowsPerImage, -1) != noErr) {
                        CleanUp(listH, &id);
                        return(false);
            }
            if (oddRowBytes)
                        TFixOddRowBytes(myBitMapPtr);
            DisplayImage(FrontWindow(), myBitMapPtr);

            CleanUp(listH, &id);
            return(true);
}

void WriteTiff(refNum, myBitMapPtr)
Int16 refNum;
BitMap *myBitMapPtr;
{
            Handle                          listH;
            Ptr                                         bufferPtr;
            Boolean                         oddRowBytes;
            Int8                            dummy;
            Int16                           byteOrder,
                                                        subfileType,
                                                        imageWidth,
                                                        imageLength,
                                                        fillOrder,
                                                        compressType,
                                                        photoInterp,
                                                        bitsPerPixel,
                                                        minSampleValue,
                                                        maxSampleValue,
                                                        orientation,
                                                        tiffRowBytes,   /* number of bytes per row in TIFF format */
                                                        plane,                                          /* dummy parameter for
TWriteImageStrip */
                                                        scrnHRes,
                                                        scrnVRes;
            Int32                           rowsPerStrip,
```

(continued on next page)

Listing One *(Listing continued, text begins on page 26.)*

```
                                                    nextFileFree,   /* next free location in output file */
                                                    startLine,
                                                    numLines,
                                                    dirOffset,
                                                    count;
        Rational        xRes,
                                                    yRes;
        Rect                            imageRect;

        /* get a handle for the in memory tag list */
        listH = NewHandle(0);
        if (MemError() != noErr) {
                        ErrorMessage(BADMEMORY);
                        return;
        }

        ScreenRes(&scrnHRes, &scrnVRes);
        imageRect = myBitMapPtr->bounds;
        /* write out 8 rows per strip - 8 is an arbitrary number */
        rowsPerStrip = MIN(imageRect.bottom - imageRect.top, 8);

        /* initialize tag values */
        byteOrder = MOTOROLA;
        subfileType = 1;
        imageWidth = imageRect.right;
        imageLength = imageRect.bottom;
        bitsPerPixel = 1;
        fillOrder = 1;
        compressType = 1;
        photoInterp = 0;
        minSampleValue = 0;
        maxSampleValue = (1 << bitsPerPixel) - 1;
        orientation = 1;
        xRes.numerator = (Int32)scrnHRes;
        xRes.denominator = 1;
        yRes.numerator = (Int32)scrnVRes;
        yRes.denominator = 1;

        tiffRowBytes = (imageRect.right * bitsPerPixel + 7) / 8;
        oddRowBytes = (tiffRowBytes % 2) != 0;

        /* Put tags in memory list.
         * The order tags are put in the list with TPutPtrTag is NOT important.
         */
        if (            TPutPtrTag(listH, SUBFILE_TYPE_TAG, SHORT,
                                                                                                1L,
&subfileType)          != noErr        ||
                                        TPutPtrTag(listH, IMAGE_WIDTH_TAG,              SHORT,
                                                                                                1L,
&imageWidth)           != noErr        ||
                                        TPutPtrTag(listH, IMAGE_LENGTH_TAG, SHORT,
                                                                                                1L,
&imageLength)          != noErr        ||
                                        TPutPtrTag(listH, ROWS_PER_STRIP_TAG, LONG,
                                                                                                1L,
&rowsPerStrip)         != noErr        ||
                                        TPutPtrTag(listH, X_RESOLUTION_TAG, RATIONAL,
                                                                                                1L, &xRes)
            != noErr    ||
                                        TPutPtrTag(listH, Y_RESOLUTION_TAG, RATIONAL,
                                                                                                1L, &yRes)
            != noErr    ||
                                        TPutPtrTag(listH, BITS_PER_SAMPLE_TAG, SHORT,
1L,&bitsPerPixel)       != noErr        ||
                                        TPutPtrTag(listH, COMPRESSION_TAG, SHORT,
                                                                                                1L,
&compressType)         != noErr        ||
                                        TPutPtrTag(listH, FILL_ORDER_TAG, SHORT,
                                                                                                1L,
&fillOrder)            != noErr        ||
                                        TPutPtrTag(listH, ORIENTATION_TAG, SHORT,
                                                                                                1L,
&orientation)          != noErr        ||
                                        TPutPtrTag(listH, PHOTOMETRIC_INTERP_TAG, SHORT,
                                                                                                1L,
&photoInterp)          != noErr        ) {
                        ErrorMessage(BADPUTTAGS);
                        return;
        }

        /* leave room for header in output file */
        SetEOF(refNum, (Int32)sizeof(TiffHeader));
        nextFileFree = sizeof(TiffHeader);

        /* Write out image to file, fixing from Macintosh rounding of rows to the
         * nearest 2 bytes, to the TIFF rounding of rows to the nearest byte.
         */
        if (oddRowBytes)
                        TUnfixOddRowBytes(myBitMapPtr);                 /* round rows to nearest byte */
```

```
            startLine = 0;
            bufferPtr = myBitMapPtr->baseAddr;
            while  (startLine < imageLength) {
                        numLines = MIN(imageLength - startLine, rowsPerStrip);
                        if (TWriteImageStrip(refNum, &nextFileFree, listH,
                                                                                    startLine, numLines, bufferPtr, plane) !=
noErr) {
                                    ErrorMessage(BADWRITEIMAGE);
                                    return;
                        }
                        startLine += numLines;
                        bufferPtr += numLines * tiffRowBytes;
            }

            if (oddRowBytes)
                        TFixOddRowBytes(myBitMapPtr);                    /* round rows to nearest word */

            /* directory must be on word boundary */
            if (nextFileFree % 2 != 0) {
                        /* add filler byte */
                        count = 1;
                        if (FSWrite(refNum, &count, &dummy) != noErr) {
                                    ErrorMessage(BADWRITE);
                                    return;
                        }
                        nextFileFree++;
            }

            dirOffset = nextFileFree;

            if (TWriteTags(refNum, byteOrder, &nextFileFree, listH, 0L) != noErr) {
                        ErrorMessage(BADWRITETAGS);
                        return;
            }
            if (TWriteHeader(refNum, dirOffset, byteOrder) != noErr) {
                        ErrorMessage(BADWRITEHEADER);
            }
}

/* Free the list handle and any memory allocated to image description structure.
 */
static void CleanUp(listHandle, idPtr)
Handle listHandle;
id *idPtr;
{
            MyDisposPtr(&idPtr->bitsPerSample);
            MyDisposPtr(&idPtr->compression);
            MyDisposPtr(&idPtr->docName);
            MyDisposPtr(&idPtr->imageDescription);
            MyDisposPtr(&idPtr->make);
            MyDisposPtr(&idPtr->model);
            MyDisposPtr(&idPtr->stripOffsets);
            MyDisposPtr(&idPtr->stripByteCounts);
            MyDisposPtr(&idPtr->minSampleValue);
            MyDisposPtr(&idPtr->maxSampleValue);
            MyDisposPtr(&idPtr->pageName);
            MyDisposPtr(&idPtr->freeOffsets);
            MyDisposPtr(&idPtr->freeByteCounts);
            MyDisposPtr(&idPtr->grayResponseCurve);
            MyDisposPtr(&idPtr->colorResponseCurves);
            DisposHandle(listHandle);
}

void InitID(idPtr)
id *idPtr;
{
            idPtr->subfileType = -1;
            idPtr->imageWidth = 0;
            idPtr->imageLength = 0;
            idPtr->bitsPerSample = nil;
            idPtr->compression = nil;
            idPtr->photoInterp = -1;
            idPtr->threshholding = -1;
            idPtr->cellWidth = -1;
            idPtr->cellLength = -1;
            idPtr->fillOrder = 0;
            idPtr->docName = nil;
            idPtr->imageDescription = nil;
            idPtr->make = nil;
            idPtr->model = nil;
            idPtr->stripOffsets = nil;
            idPtr->orientation = -1;
            idPtr->samplesPerPixel = 0;
            idPtr->rowsPerStrip = 0;
            idPtr->stripsPerImage = 0;
            idPtr->stripByteCounts = nil;
            idPtr->minSampleValue = nil;
            idPtr->maxSampleValue = nil;
            idPtr->xResolution.numerator = 0;
            idPtr->xResolution.denominator = 0;
            idPtr->yResolution.numerator = 0;
            idPtr->yResolution.denominator = 0;
            idPtr->planarConfig = -1;
            idPtr->pageName = nil;
```

(continued on next page)

Listing One *(Listing continued, text begins on page 26.)*

```
            idPtr->xPosition.numerator = 0;
            idPtr->xPosition.denominator = 0;
            idPtr->yPosition.numerator = 0;
            idPtr->yPosition.denominator = 0;
            idPtr->freeOffsets = nil;
            idPtr->freeByteCounts = nil;
            idPtr->grayResponseUnit = -1;
            idPtr->grayResponseCurve = nil;
            idPtr->group3Options = 0;
            idPtr->group4Options = 0;
            idPtr->resolutionUnit = -1;
            idPtr->pageNumber[0] = 0;
            idPtr->pageNumber[1] = 0;
            idPtr->colorResponseUnit = -1;
            idPtr->colorResponseCurves = nil;
}

TiffDirEntry GetDirEntry(listHandle, tagOffset)
Handle      listHandle;
Int32       tagOffset;
{
            Ptr                                                     p;

            /* get pointer to tag list */
            p = &(**listHandle);            /* HANDLE DEREFERENCE */
            /* get pointer to our tag's directory entry */
            p += tagOffset;
            /* return the whole Directory Entry Structure, not a pointer to it */
            return(*((TiffDirEntry *)p));
}


Int16 TypeSize(type)
Int16 type;
{
            switch (type) {
                        case BYTE:
                                    return(BYTESIZE);
                        case ASCII:
                                    return(ASCIISIZE);
                        case SHORT:
                                    return(SHORTSIZE);
                        case LONG:
                                    return(LONGSIZE);
                        case RATSIZE:
                                    return(RATSIZE);
            }
}

Ptr SetIDPtr(listHandle, tag, defaultValue, defaultType)
Handle listHandle;
Int16 tag;
Int32 defaultValue;                 /* won't handle defaults larger than 4 bytes */
Int32 defaultType;
{
            TiffDirEntry    tagDE;
            Ptr                                                     p;
            Int32                                       tagOffset,
                                                                    size,
                                                                    nvals;

            if (TFindTag(listHandle, &tagOffset, tag)) {
                        tagDE = GetDirEntry(listHandle, tagOffset);
                        size = TypeSize(tagDE.type) * tagDE.length;
                        if ((p = MyNewPtr(size)) != nil)
                                    TGetTag(listHandle, tagOffset, p, size);
            }
            else if (defaultType != 0) {
                        if ((p = MyNewPtr(defaultType)) != nil) {
                                    switch (defaultType) {
                                                case BYTE:
                                                            *(unsigned char *)p = defaultValue;
                                                            break;
                                                case ASCII:
                                                            *(unsigned char *)p = defaultValue;
                                                            break;
                                                case SHORT:
                                                            *(unsigned short *)p = defaultValue;
                                                            break;
                                                case LONG:
                                                            *(unsigned long *)p = defaultValue;
                                                            break;
                                                case RATIONAL:
                                                            ((Rational *)p)->numerator = defaultValue;
                                                            ((Rational *)p)->denominator = 1;
                                                            break;
                                    }
                                    }
                        }
                        else
                                    p = nil;
                        return(p);
            }
```

End Listings

Listing One *(Text begins on page 46.)*

```
Listing One

/* Virtual Array Header File */
#include <alloc.h>
#include <stdio.h>

/* Virtual Array Control Block typedef */

typedef struct {

   FILE *file;            /* pointer to file control block */
   long size;             /* number of array elements in file */
   int  elsize;           /* number of bytes in each element */
   char *buffer;          /* pointer to array buffer */
   int  buf_elsize;       /* size of element in buffer including index */
   int  buf_size;         /* number of elements in buffer */
   char *blank_rec;       /* pointer to initialization record */
                          /* used for extending file */
} VACB ;                  /* virtual array control block type name */

/* Virtual Array Access Prototypes */

int init_v_array(char *filename,int rec_size,char filchar);
VACB *open_v_array(char *filename,int buffer_size);
void close_v_array(VACB *v_array);
void *access_v_rec(VACB *v_array,long index);
```

End Listing One

Listing Two

```
Listing Two

/*  Virtual Array Access Routines  */

#include <varray.h>

#define header 7

/*****************
 * init_v_array *
 *****************/

int init_v_array(filename,rec_size,filchar)
char *filename, filchar;
int  rec_size;
{
   long size;
   FILE *f;
   f = fopen(filename,"wb");
   if (f != NULL) {
      size = 0;
      fwrite(&size,4,1,f);          /* write array size of zero */
      fwrite(&rec_size,2,1,f);      /* and array element size   */
      fwrite(&filchar,1,1,f);       /* and fill char            */
      fclose(f);                    /* to file header */
      return(1);
   }
   else
      return(NULL);
}

/*****************
 * open_v_array *
 *****************/

VACB *open_v_array(filename,buffer_size)
char *filename;
int  buffer_size;
{
   VACB *v_array;
   char *buf_ptr;
   int  i;
   char filchar;

   /* allocate virtual array control block */

   v_array = (VACB *) malloc(sizeof(VACB));
   if (v_array == NULL) return(NULL);

   /* open virtual array file */

   v_array->file = fopen(filename,"r+b");
   if (v_array->file == NULL) {
      free(v_array);
      return(NULL);
   };

   /* get array size and element size for control block */

   fread(&v_array->size,4,1,v_array->file);
   fread(&v_array->elsize,2,1,v_array->file);
   fread(&filchar,1,1,v_array->file);
   v_array->buf_elsize = v_array->elsize + 4;


   /* allocate buffer */

   v_array->buffer = (char *) malloc(v_array->buf_elsize * (buffer_size + 1));
   if (v_array->buffer == NULL) {
      fclose(v_array->file);
      free(v_array);
      return(NULL);
   };
   v_array->buf_size = buffer_size;

   /* set up blank_rec using the fill character in array header */
   /* for initializing new array elements */

   buf_ptr = v_array->buffer + v_array->buf_elsize * v_array->buf_size;
   v_array->blank_rec = buf_ptr + 4;
   for (i=0; i < v_array->buf_elsize; i++)
      *buf_ptr++ = filchar;

   /* set record index negative for all buffer elements   */

   buf_ptr = v_array->buffer;
   for (i = 0; i < v_array->buf_size; i++) {
      *((long *)buf_ptr) = -1L;
      buf_ptr += v_array->buf_elsize;
   };
   return(v_array);
}

   /*****************
    * close_v_array *
    *****************/

   void close_v_array(v_array)
   VACB *v_array;
   {
      int  i;
      char *buf_ptr;
      long rec_index, file_offset;

      buf_ptr = v_array->buffer;

      /*  flush buffer */

      for (i=0; i < v_array->buf_size; i++) {

         /* check each element index */
         /* if element present, write it to disk */

         rec_index = *((long *)buf_ptr);
         if (rec_index >= 0) {
               file_offset = header + rec_index * v_array->elsize;
               fseek(v_array->file,file_offset,0);
               fwrite(buf_ptr + 4, v_array->elsize,1, v_array->file);
         };
         buf_ptr += v_array->buf_elsize;
      };
      free(v_array->buffer);      /* de-allocate buffer */
      fclose(v_array->file);      /* close array file   */
      free(v_array);              /* de-allocate VACB   */
   }

   /*****************
    * access_v_rec *
    *****************/

   void *access_v_rec(v_array,index)
   VACB *v_array;
   long index;
   {
      char *buf_ptr;
      int  buf_index;
      long rec_index, temp_index;

      /* calculate buffer address of referenced element */

      buf_index = index % v_array->buf_size;
      buf_ptr = v_array->buffer + buf_index * v_array->buf_elsize;
      rec_index = *(long *)buf_ptr;

      /* if element present, return its buffer address */

      if (rec_index == index) return(buf_ptr + 4);

      /* if element doesn't exist, extend the file */

      if (index >= v_array->size) {
         fseek(v_array->file,0,2);
         for (temp_index = v_array->size; temp_index++ <= index; )
               fwrite(v_array->blank_rec, v_array->elsize, 1, v_array->file);
         v_array->size = index + 1;
         fseek(v_array->file,0,0);
         fwrite(&v_array->size, 4, 1, v_array->file);
      };

   /* if buffer slot is occupied by another element, */
   /* save it to disk */

   if (rec_index >= 0) {
      fseek(v_array->file, rec_index * v_array->elsize + header, 0);
      fwrite(buf_ptr + 4, v_array->elsize, 1, v_array->file);
   };

   /* read referenced element into buffer slot */

   fseek(v_array->file, index * v_array->elsize + header, 0);
   fread(buf_ptr + 4, v_array->elsize, 1, v_array->file);
   *((long *)buf_ptr) = index;

   /* return address of element in buffer */

   return(buf_ptr + 4);
}



Listing Three

/* Example Program Using Virtual Arrays */

#include <varray.h>

/* Access Macros */

#define VREC(i) ((items *)access_v_rec(item_array,i))
#define item(i) VREC(i)->v_item
#define qty(i) VREC(i)->v_qty
#define desc(i) VREC(i)->v_desc

/* Array element structure typedef */

typedef struct {
   int v_item,
       v_qty;
   char v_desc[24];
} items;
```

VIRTUAL ARRAYS

Listing Two *(Listing continued, text begins on page 46.)*

```
main()
{
   VACB *item_array;
   long i;

   /* create a virtual array setting element size to */
   /* the size of items structure and setting the    */
   /* initialization char to the space char          */

   init_v_array("ITEMS.VAR",sizeof(items),' ');

   /* open the virtual array, reserve buffer space for 10 elements */

   item_array = open_v_array("ITEMS.VAR",10);


   /* create 50 array items */

   for (i = 0; i < 50 ; i++) {
      item(i) = i + 1;
      qty(i)  = 0;
      sprintf(desc(i)," Item # %ld", i + 1);
   }

   /* print contents of the 50 array items */
   /* plus the ascii code of last char in v_desc */

   for (i = 0; i < 50; i++)
      printf("Element # %ld  Item = %d  Qty = %d Desc = %s  %d\n",
            i, item(i), qty(i), desc(i), (int) desc(i)[23]);

   /* close virtual array */

   close_v_array(item_array);
}
```

End Listings

C CHEST

Listing One *(Text begins on page 72.)*

```
#include <ctype.h>

char *factor( str )
char    *str;
{
    char *expr();

    if( isalpha( *str ) )       /* F -> name  */
        printf( "%c\n", *str );
    else if( *str == '(' )      /* F -> ( E ) */
        str = expr( ++str );
    return ++str;
}

char *term( str )
char    *str;
{
    str = factor( str );        /* T -> FT'  */
    while( *str == '*' )        /* T'-> *FT' */
    {
        str++;
        str = factor( str );
        printf( "*\n" );
    }
    return str;                 /* T'-> eps */
}

char   *expr( str )
char   *str;
{
    str = term( str );          /* E -> TE'  */
    while( *str == '+' )        /* E'-> +TE' */
    {
        str++;
        str = term( str );
        printf( "+\n" );
    }
    return str;                 /* E'-> eps */
}

main()
{
    char  buf[80];
    while( gets(buf) )
        expr( buf );
}
```

End Listing One

Listing Two

```
typedef struct node
{
    char        name[16];
    int         op;
    struct node *left;
    struct node *right;
}
NODE;

NODE *new()
{
    NODE *p;
    void *calloc();

    if( !(p = (NODE *) calloc( 1, sizeof(NODE) ) ) )
        exit(1);

    return p;
}
```

End Listing Two

Listing Three

```
NODE    *build()
{
    char   buf[80];
    NODE   *stack[ 10 ];
    NODE   **sp = stack - 1;
    NODE   *p;

    while( gets(buf) )
    {
        switch( *buf )
        {
        default:  p = new();
                  strcpy( p->name, buf );
                  *++sp = p;
                  break;

        case '*':
        case '+':
                  p          = new();
                  p->right   = *sp-- ;
                  p->left    = *sp-- ;
                  p->op      = *buf  ;
                  p->name[0] = *buf  ;
                  *++sp      = p     ;
                  break;
        }
    }
    return *sp--;
}
```

End Listing Three

Listing Four

```
trav( root )
struct node *root;
{
    static int tnum = 0;

    if( !root )
        return;

    if( !root->op )     /* leaf */
    {
        printf ( "t%d = %s\n", tnum, root->name );
        sprintf( root->name , "t%d", tnum );
        ++tnum;
    }
    else
    {
        trav( root->left  );

        if( root->left != root->right )    /* Always true      */
            trav( root->right );           /* unless optimized */

        printf("%s %c= %s\n", root->right->name,
                              root->op, root->left->name );
        strcpy( root->name,   root->right->name );
    }
}
```

End Listing Four

Listing Five

```
optimize( root )
NODE    *root;
{
    /* Stupid optimizer to eliminate common subexpressions */

    char sig1[ 32 ];
    char sig2[ 32 ];

    if( root->right && root->left )
    {
        optimize( root->right );
        optimize( root->left  );

        *sig1 = *sig2 = '\0';
        makesig( root->right, sig1 );
        makesig( root->left , sig2 );

        if( strcmp( sig1, sig2 ) == 0 )   /* subtrees match */
            root->right = root->left ;
    }
}

makesig( root, str )
NODE    *root;

        char    *str;
        {
            if( !root )
                return;

            strcat( str, root->name );
            makesig( root->left,  str );
            makesig( root->right, str );
        }
```

End Listings

Listing One *(Text begins on page 92.)*

```
Listing One

/* MKTABLE.C: Makes a data table with header record */

#include <stdio.h>
#include <string.h>
#define  SIG  19364            /* application file signature */

typedef struct {
  char      fname [20];
  int       ftype, flen;
} DESCR;

struct {                              /* header record for file */
  unsigned  signature;
  int       nrecs;
  char      tablename [10];
  int       reclen;
  long      datastart;
  int       descrsize;
  int       ndescr;
} header;

struct {                               /* data record for file */
  char      name [20];
  int       age;
} data;

main ()
{
FILE  *fp;
char  age [3];
int   n;
DESCR descr;

  fp = fopen ("database.xyz", "w");           /* create file */

  header.signature = SIG;                /* initialize header */
  header.nrecs = 0;
  strcpy (header.tablename, "Age list");
  header.reclen = sizeof data;
  header.datastart = 256L;
  header.descrsize = sizeof (descr);
  header.ndescr = 2;
  fwrite (&header, sizeof header, 1, fp);   /* write to file */

  strcpy (descr.fname, "NAME");      /* initialize descriptor */
  descr.ftype = 1;
  descr.flen  = 20;
  fwrite (&descr, sizeof (descr), 1, fp);   /* write to file */

  strcpy (descr.fname, "AGE");                /* ditto above */
  descr.ftype = 0;
  descr.flen  = 2;
  fwrite (&descr, sizeof (descr), 1, fp);

  fseek (fp, 256L, SEEK_SET);

  do {                                       /* capture data */
    printf ("\nName? ");
    gets (data.name);
    if (strlen (data.name)) {         /* continue until blank */
      printf ("Age?  ");
      gets (age);
      data.age = atoi (age);
      fwrite (&data, sizeof data, 1, fp);    /* write record */
      header.nrecs += 1;                     /* count record */
    }
  } while (strlen (data.name));      /* until no more entered */

  fseek (fp, 0L, SEEK_SET);           /* go to start of file */
  fwrite (&header, sizeof header, 1, fp);   /* update header */
  fclose (fp);                              /* close file */
}
```

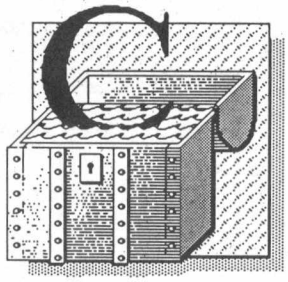
End Listing One

Listing Two

```
Listing Two

PROGRAM nonpas;

  { Reads a non-Pascal database table with a header record }
  { and some number of fixed-length data records           }

CONST signature = 19364;               { application signature }
      divider = '-------------------------------------------';

TYPE  s20             = STRING [20];
      pac             = PACKED ARRAY [1..20] OF CHAR;

  headrec = RECORD CASE tag : INTEGER OF
  1: (signature  : WORD;                  { This is the real layout }
      nrecs      : WORD;                                { # data records }
      placeholdr : PACKED ARRAY [1..10] OF CHAR;    { table name }
      reclen     : INTEGER;                  { data record length }
      datastart  : LONGINT;                { file offset for data }
      descrsize  : INTEGER;               { field descriptor size }
      ndescr     : INTEGER);           { number of fields per rec }
  2: (dummy1,
      dummy2     : WORD;
      tablename  : pac);                    { To fool typechecking }
  3: (stream     : PACKED ARRAY [1..24] OF BYTE);
  END;

  fieldrec = RECORD CASE tag : INTEGER OF
  1: (fname      : pac;
      ftype      : INTEGER;
      flen       : INTEGER);
  2: (stream     : PACKED ARRAY [1..24] OF BYTE);
  END;

VAR   header   : headrec;
      field    : ARRAY [1..10] OF fieldrec;       { descriptors }
      n        : INTEGER;
      table    : FILE OF BYTE;
{ --------------------------- }

FUNCTION asciiz (max : INTEGER; VAR strng : pac) : s20;

    { Returns a Pascal string from a null-terminated string
      that is <= max bytes long }

VAR   i      : INTEGER;
      result : STRING [20];

BEGIN
  result := '';
  FOR i := 1 TO max DO
    IF strng [i] <> CHR (0) THEN
    result := result + strng [i];
  asciiz := result;
END;
{ --------------------------- }

PROCEDURE getDescriptors;

    { Reads field descriptors from header record }

VAR   c, d : INTEGER;

BEGIN
  FOR d := 1 to header.ndescr DO
    FOR c := 1 TO header.descrsize DO
      READ (table, field [d].stream [c]);
END;
{ --------------------------- }

PROCEDURE showHeaderInfo;

    { List information about the file format }

VAR   d : INTEGER;

BEGIN
  WRITELN (divider);
  WRITELN ('Table name is ',
           asciiz (10, header.tablename));
  WRITELN ('Table contains ', header.nrecs, ' records');
  WRITELN ('Data record length in bytes is ',
           header.reclen);
  WRITELN ('Each record contains ', header.ndescr, ' fields:');
  getDescriptors;
  FOR d := 1 TO header.ndescr DO BEGIN
    WRITELN ('  Field name:    ', asciiz (20, field [d].fname));
    WRITE   ('  Data type:     ');
    CASE field [d].ftype OF
      0: WRITELN ('Integer');
      1: WRITELN ('Character');

        END;
        WRITELN ('  Length:        ', field [d].flen);
        WRITELN;

  END;
  WRITELN ('Data records follow:');
  WRITELN;
END;
{ --------------------------- }

PROCEDURE showData;

      { List contents of each data record by fieldname }

TYPE  int = RECORD CASE tag : INTEGER OF
        1: (number : INTEGER);
        2: (stream : PACKED ARRAY [1..2] OF BYTE);
      END;

TYPE  charfield = RECORD CASE tag : INTEGER OF
        1: (bf : PACKED ARRAY [1..20] OF BYTE);
        2: (cf : pac);
      END;

VAR   rec, descr, n : INTEGER;
      intfield      : int;                  { integer data field }
      chfield       : charfield;          { character data field }

BEGIN
  FOR rec := 1 TO header.nrecs DO              { For each record }
    FOR descr := 1 TO header.ndescr DO BEGIN   { For each field }
      WRITE (asciiz (20, field [descr].fname));      { Show name }
      FOR n := LENGTH (asciiz (20, field [descr].fname)) TO 25 DO
        WRITE (' ');                                { cosmetic spacing }
      CASE field [descr].ftype OF
        0: BEGIN
        FOR n := 1 TO 2 DO
          READ (table, intfield.stream [n]);    { get int field }
        WRITELN (intfield.number);
      END;
        1: BEGIN
        FOR n := 1 TO field [descr].flen DO
          READ (table, chfield.bf [n]);   { get character field }
        WRITELN (asciiz (20, chfield.cf));
      END;
      END;
    END;
END;
{ --------------------------- }

BEGIN
  ASSIGN (table, 'DATABASE.XYZ');                  { open table }
  RESET (table);
  FOR n := 1 TO 24 DO                          { read header record }
    READ (table, header.stream [n]);
  IF signature <> header.signature THEN
    WRITELN ('File not in proper format. Program ended.')
  ELSE
    BEGIN
      showHeaderInfo;                 { Show info about the file }
      SEEK (table, header.datastart);      { go to start of data }
      showData;                          { List each record's data }
    END;
  CLOSE (table);
END.
```

End Listings

Postfix Notation and Common-Subexpression Elimination



Most compilers don't generate binary executable code; rather, they create a program in an "intermediate language" that is translated into binary by a "compiler back end." This approach has three advantages. First, the same front end can generate code for many target machines by providing several back ends. Second, and conversely, compilers for several different languages can generate the same intermediate code, which can then be used by a single back end. And finally, intermediate code is usually easier to optimize than assembly language is.

It's the third advantage that's the main topic of this month's column: a small compiler along with a simple optimizer that does common-subexpression elimination. The code this month is not all that useful as it stands. For one thing I've left out most of the error checking to make it more understandable. Nonetheless, the concepts are quite useful and apply to several applications apart from compiler design.

One common intermediate language that compilers use is a postfix or reverse-Polish notation. Users of Hewlett-Packard calculators and Unix's dc desk-calculator program will already be familiar with the process. In postfix notation, operands are pushed onto a stack without modification. Operators, however, modify the top few items on the stack. The C fragment A*B+A*B, for example, generates the following postfix operations:

by Allen Holub

```
push A
push B
pop two items, multiply them, push
                                the result
push A
push B
pop two items, multiply them, push
                                the result
pop two items, add them, push the
                                result
```

When the input is processed, the result will be on the top of the stack.

A postfix intermediate language is easy to generate because the compiler doesn't have to worry about assigning temporary variables for the rvalues; it just uses the stack as its scratch space. The second advantage is that the optimization pass can reconstruct the entire parse tree—or to be more exact, a compacted form of the parse tree called a syntax tree—from the list of instructions. It turns out that several optimizations are much easier to perform on a syntax tree than on a quad representation of the same program.

Let's look at a concrete example. The earlier expression *(A*B)+(A*B)* can be represented in the following postfix form:

```
A
B
*
A
B
*
+
```

There's no need for an explicit *push* operator as long as the operators can be distinguished from variable names. Similarly, explicit parentheses are never necessary because the order of evaluation is determined by the sequence of operations.

The set of subroutines in Listing 1, page 67, form a small recursive-decent compiler that takes as input expressions involving single-character variable names, plus signs, times signs (asterisks), and parentheses. The program outputs the input expression in postfix notation. Multiplication is of higher precedence than addition, and expressions associate left to right. Given the expression *(A*B)+(A*B)*, it produces the output described earlier. The grammar is shown in Table 1, below.

The next step is to recreate the syntax tree from the postfix expression output by the parser (see Figure 1, page 73). Note that all internal nodes represent operators and all the leaves are variable references. Also, the grouping of operators and operands is as you would expect, given the original parenthesized expressions, even though there were no parentheses in the postfix expression itself.

Before demonstrating how to do this reconstruction, I need a data structure to represent the nodes in the tree. Listing 2, page 67, shows this data structure—*NODE*—and a constructor subroutine that makes new nodes—*new( )*. The *NODE* structure is a normal binary-tree node,

expr	<– term expr'	Implemented in expr( )
expr	<– + term expr'	"
	<– ϵ	"
term	<– factor term'	Implemented in term( )
term'	<– * factor term'	"
	<– ϵ	"
factor	<– **name**	Implemented in factor( )
	<– (expr)	"

Table 1: *Grammar used by a postfix compiler*

having left and right children. In addition, the *name* field holds variable names (*A* and *B* in this case) or the operator if the node is an internal node. (The contents of this field will be modified by the optimizer, however.) The *op* field holds the operator (* or +) or is set to 0 on leaf nodes.

The *build( )* subroutine, shown in Listing 3, page 67, creates a syntax tree from a postfix input file generated by the parser I just discussed. The input file must have one operand or operator per line, and it must be perfect—that is, in order to simplify the code, I've dispensed with error detection. Input lines are read from standard input, and the subroutine returns a pointer to the root node of the tree.

Trees are built in a bottom-up fashion, using a local stack to keep track of the partially constructed tree. The *default* case is executed for variable names. It allocates and initializes a new node and then pushes a pointer to the new node onto the stack. The child pointers are initialized to *NULL* by *new( )*.

Operators are handled differently because they're internal nodes. A new node is allocated and initialized, then pointers to two existing nodes are popped and the child pointers of the new internal node are made to point at these. Finally, the new node is pushed.

The tree for the input discussed earlier is built as shown in Figure 2, this page. Code can be generated from this tree by doing a depth-first traversal (visit the children, then the parent). At every lvalue (that is, variable reference), generate a temporary=variable instruction. At every internal node, generate the code necessary to perform the operation on the temporaries that resulted from traversing the previous level, putting the result into a new temporary.

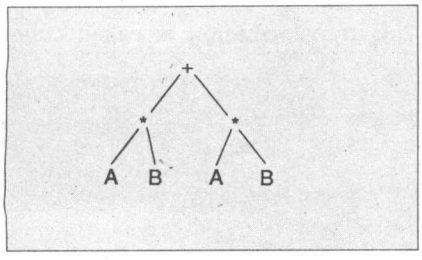


Figure 1: *The syntax tree*

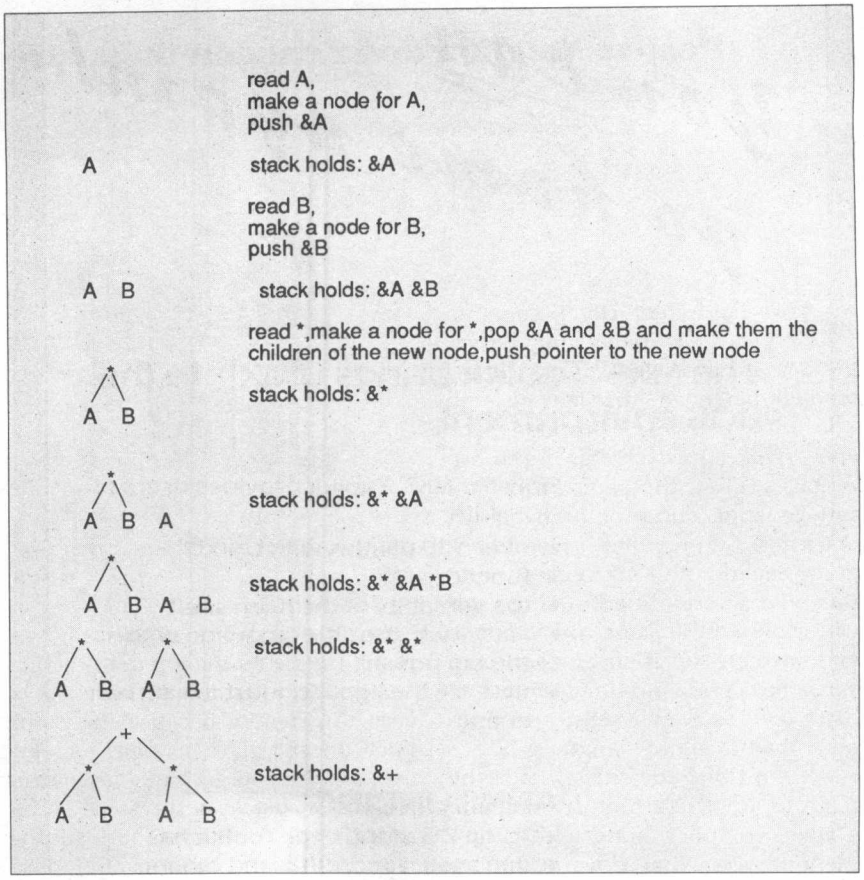


Figure 2: *The input tree*

That is, the previously constructed tree will generate the following output:

```
t0 = A
t1 = B
t1 *= t0
t2 = A
t3 = B
t3 *= t2
t3 += t1
```

The *trav( )* subroutine in Listing 4, page 67, does the traversal. It takes the pointer returned from the previous *build( )* call as its initial argument. If *root->op* is zero, then the current node is a leaf, and you generate the code to move it to a temporary variable. The *sprintf( )* call overwrites the *name* field with the name of the temporary variable. If the *op* field is nonnull, you're processing an interior node. In this case, you do an in-order traversal. The *if* statement is always true (for now—things will change momentarily). The following *printf( )* call prints the instruction, using the *name* fields of the two children to find out what temporaries to use. The *strcpy( )* call then overwrites the *name* field of the current node to reflect the temporary that got the result of the last operation.

The code that *trav( )* outputs isn't too great, primarily because the subexpression *A*B* is evaluated twice. It would be better to perform the multiply only once and use the temporary rvalue generated by that multiply twice. That is, you'd like to generate the following output:

```
t0 = A
t1 = B
t1 *= t0
t1 += t1
```

This transformation is called *com-*

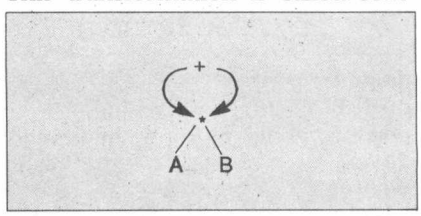


Figure 3: *The new syntax tree*

Flotsam and Jetsam

More Rvalues and Implicit Type Conversion

This month's Flotsam and Jetsam continues the subject of rvalues started last month by looking at another problem caused by rvalues and the automatic type-conversion rules. The following code does not work properly:

```
long x;
int y = 32767, z = 32767;
x = y * z;
```

If your machine has a 16-bit *int*, *x* will have the value 1 assigned to it.

The problem here is rvalues—temporary variables that are used by the compiler to hold the intermediate steps of an expression evaluation. The compiler interprets the previous code using the following steps:

1. Copies *y* to the *int*-size temporary *t0*
2. Copies *z* to the *int*-size temporary *t1*
3. Multiplies *t0* by *t1* and puts the result into the *int*-size temporary *t2*
4. Converts *t2* to *long* and puts the result into the *long*-size temporary *t3*
5. Copies *t3* to *x*

Because the result of the multiplication is stored in an *int*-size temporary, the product will be truncated. That is, 32,767 * 32,767 should yield 0x3fff0001, the top four digits of which will be truncated as part of the assignment.

The problem is that the compiler has no general understanding of the expression as a whole; rather, it knows only about a single operator and the associated operands. It breaks up the foregoing expression into *(x = (y * z))* and then starts evaluating from the inside out—that is, it starts with the multiplication. Because *y* and *z* are both of type *int*, the result of evaluating the subexpression is put into an *int*-size rvalue. Only after the multiplication is performed will the compiler move out a nesting level and apply the = operator. Now the operands will be *x* and the rvalue that resulted from the previous subexpression. Noticing that *x* is of type *long* and that the temporary is of type *int*, the compiler will correctly convert the *int* to *long* before doing the assignment.

All this is not a bug—the compiler is just doing what you told it to do. You can fix the problem in several ways. The easiest is to declare either *x* or *y* (or both) as *long*. This way the conversion to *long* is moved up to an earlier stage of the evaluation process. Assuming that you've made *y* a *long*, the compiler will do the following:

1. Copy *y* to the *long*-size temporary *t0*
2. Copy *z* to the *int*-size temporary *t1*
3. Noticing that the two operands are of different types, convert the *int*-size *t1* to the *long*-size temporary *t2*
4. Multiply *t0* by *t2* and put the result into the *long*-size temporary *t3*
5. Copy *t3* to *x*

Alternatively, you could use the cast operator, which creates a temporary variable that has the specified type and then copies the operand to the new variable. You can use any of the following:

```
x = (long)y * z;
x = y * (long)z;
x = (long)y * (long)z;
```

Remember, here, that the cast is an operator. That is, it's executed at run time (unless the optimizer can fix things), and it creates an rvalue of a given type and copies something into that rvalue, doing any necessary type conversions along the way.

You form a cast by writing out a declaration of a variable having the desired type, surrounding the declaration with parentheses, and removing the name and semicolon. For example, to create a cast for a pointer to an array of structures of type *building*, you can use the following procedure:

```
struct building (*p)[10];
(struct building (*p)[10];)
(struct building (*)[10])
```

Note that the parentheses around **p* are required here because *p* is a pointer to an array of structures (as compared to an array of pointers to structures). The parentheses are needed because the brackets are of higher precedence than the start, so the default binding (without parentheses) is *struct building *(p[10])*;.

—*A.H.*

mon-subexpression elimination. The optimization is performed by analyzing and then modifying the syntax tree. Because both subtrees of the + node are identical, the optimizer can eliminate one subtree and make both pointers in the + node point at the remaining subtree. The new syntax tree looks like that in Figure 3, page 74. That is, both pointers in the + node point at the * node. This modified data structure is called a *directed acyclic graph*, or *DAG*.

The DAG is created from the syntax tree by the *optimize( )* function in Listing 5, page 68. This routine traverses the interior nodes of the tree, comparing the two subtrees. If the subtrees are identical, the left and right pointers of the parent node are made to point at the same child, effectively removing the other child from the tree. The comparison is done using the *makesig( )* function, which traverses an entire subtree, assembling a string that shows the in-order traversal of the subtree by concatenating all the *name* fields. For example, the original syntax tree, when traversed from the root, creates the signature string *+**ABAB*. If two subtrees generate the same signature, they're equivalent.

If two subtrees generate the same signature, they're equivalent.

Finally, you can traverse the DAG using the same *trav( )* function that you used earlier. That *if* statement will now come into play, however, preventing you from traversing the common subtree twice.

Bibliography

This month's column is an excerpt from a forthcoming book (by myself) on compiler design, to be published by Prentice-Hall in late 1988 (or early 1989).

You can also refer to the following books:

Aho, Sethi, and Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, 1986. Pages 25-82 have a discussion about recursive-decent parsing, and pages 279-387 contain more information about advanced optimization techniques using DAGs.

Gries, David. *Compiler Construction for Digital Computers*. New York: Wiley, 1971: 376-411. This book also has a discussion about recursive-descent parsing.

Holub, Allen. *The C Companion*. Englewood Cliffs, N.J.: Prentice-Hall, 1987: 189-212. This book contains an introductory-level discussion of recursion in general and recursive-decent parsers in particular.

Sarff, Gary. "Optimization Strategies." *Computer Language*, vol. 2, no. 12 (December 1985): 27-32. This article has a good discussion of simple peephole-optimization techniques.

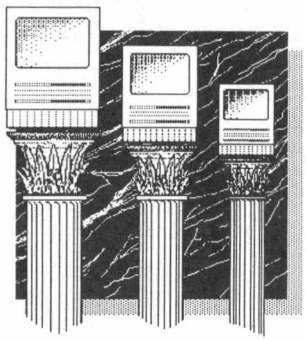
DDJ

(Listings begin on page 67.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

TO THE MACS

A Macworld Expo Potpourri and the Scouting Toolkit Wrap-Up



I almost didn't go to January's Macworld Expo in San Francisco. I was busy, I was tired, I could read about it later, there was so much to keep learning right here at home, various project deadlines were screaming, late 20th century cities are filled with nuts, it was already too late for one of the really good parties, and on and on—the typical list of yattas. But something inside whispered, "Go." I pay attention to that voice; otherwise it slaps me silly. I went.

Expo in a Nutshell

Quick bottom line: The Mac has won, and this Expo was the happy celebration.

Reasons for victory? Hmm.... How about snap and sizzle? It just feels good to use a Mac. There's some great Mac software. The hardware's gained a lot of horsepower. Thanks go to a lot of people: Xerox PARC, the original Apple Mac team, the early buyers, Apple's tech writers and support teams, all the programmers who've survived the steep learning curve, David E. Smith and his stable of MacTutor writers, Motorola's 680x0 chip design teams, all those who opened the hardware, Jobs, Sculley, and on and on beyond the spatial limits of this column.

The crowd energy at the Expo was a big wave, with as fine a glassy-walled tube to hang a cybernetic ten within as any mass computer event I've attended. There was too much to see, too many people to talk to,

by Stan Krute

and some fun parties with good talk and/or loud, danceable music. All in all, a mighty fine fourth birthday.

Hackus Triumphus

The Mac made one of its first big public appearances at the 1984 West Coast Computer Faire. While talking to an Apple representative at that event, I yammered about memory, hard disks, slotlessness, and printer limitations. He scanned me up and down. Our disguises are easily pierced. "You're a hacker. Those are just hacker issues. Normal users don't need that stuff. We're giving them what they need."

So the Mac took off fast, then almost died. More RAM, SCSI drives, the Mac II, and the LaserWriter, in synergy with some innovative third-party software, brought it back to life and success. One of the treats of the Expo was seeing so many of the people who contributed to that triumph, the once-maligned hackers, now sitting in decision-making positions at Apple and the third-party development companies. Apple's in a particular hacker-hiring frenzy these days, picking up some great Mac people at a torrid pace. Sweet success over time is the best retort to silliness, eh?

Your Code, Your Skills

Looking for places other than Apple to sell all those great Mac programs and programming skills you're busy developing? As I roamed the Expo, I continually asked company representatives if they were looking to buy. I'm not your humble servant for nothing. They were. The consensus: Competent Mac programmers are still a rare commodity. Especially ones who are able/willing to write robust programs that provide specific solutions in very Mac-like ways and don't mind reworking their code in response to intelligent critiques.

So get copies of the Expo guide and/or the leading magazines so you'll have some addresses, crank up the word processor and mail-merge software, and send out those inquiry letters—the window won't be open this wide forever.

Buy This Book

Scott Knaster was at the Expo. He's the Mac tech-support whiz who left Apple for a start-up stint with Guy Kawasaki at Acius. He's now back at Apple, deep into future stuff, and has a new book out: *Macintosh Programming Secrets*, published by Addison-Wesley.

Scott's been immersed in Mac programming for a long time and knows a *lot* about the art and science of it. There may be someone who's examined and debugged more Mac source code from more different programmers than Scott has, but I doubt it. His book pulls together a wide range of important, state-of-the-art, Macintosh programming concepts. It's got grand overviews, detailed techniques, and useful summaries. A lot of stuff you just won't see anywhere else. Plus, Scott explains things well. His writing style's smooth. He throws in history, cartoons, funny jokes, and even a special cameo appearance by Our Nation's Leader.

Sometimes, usually around 3:00 in the morning, I think I know what's going on inside the Mac. The thought has a short half-life. This book gives a more permanent effect and works in the bright light of day. It's one of those products that'll let you enjoy getting smarter. Back in the January column, I gave a short list of books that form the core of a good Mac programming library. Add this one to the group, near the top.

News from APDA

I wanted to talk with Dave Lingwood and Dick Hubert, a couple of

longtime Apple communitarians and leaders, respectively, of the Apple Programmer's and Developer's Association and its mother ship, the A.P.P.L.E. Co-op. But they were always in the midst of a continuous feeding frenzy at the APDA booth.

Luckily, I bumped into Frank Catalano, the knowledgeable APDA public relations manager. APDA's up to around 20,000 members. It's got its typical order turnaround time down from 5–7 days to 48 hours. *APDAlog*, the quarterly newsletter/catalog, keeps improving. I've got the January 1988 issue in front of me, and it's fun: good articles that are worth reading and lots of new products to slaver over, including final MPW 2.0.2 tools, a LaserWriter IISC reference, and 14 pages of third-party software tools and books. MPW's pretty remarkable stuff. (Yep, I've bitten the bullet and started to dive in; reports forthcoming, as I always seem to say.)

Of special interest to HyperTalkers: APDA's *HyperCard Technical Reference*, mentioned in the March column, has been enlarged, modified, and split into two pieces. New piece number 1, *HyperCard Script Language Guide*, is Apple's definitive 200-page Hypertalk reference manual. This is a final APDA draft, about 50 percent larger than the earlier (August 11, 1987) draft. New piece number 2, *HyperCard Developer's Toolkit*, contains several goodies. There's a nice typeset document, *HyperCard Stack Design Guidelines*, from Apple's wide-awake Human Interface Group. It lays out the basic principles of intelligent stack design in a coherent way. There's a long MacWrite document that provides the details of writing XCMDs and XFCNs, along with several very useful examples in C and Pascal. There are five sample stacks that illustrate using the included XCMDs to cope with serial-port communications, sounds, and videodisc players. Finally, they've tossed in three sounds: a clang, a flute, and a weird voice saying, "Hi, there." If you're serious about HyperCard development, you'll find both these packages vital.

Finally, if you've got a book or software tool that'd be of interest to other Mac programmers, send it to APDA for possible inclusion in *APDAlog*. APDA scans carefully, and the good stuff gets in. Not that any of us think of money (heh heh), but you could do a lot worse than to land something in this highly targeted publication.

Tools Keep Evolving, As Do Their Users

Yaaarrrggghh, always running out of time and space. Good ol' Tyler'll have my head. But, before I get down to some code talk: I was able to speak to representatives from Apple, Borland, Think (now of Symantec), Coral, and Smethers Barnes about all their hot new programming tools. Latest versions have hit the streets, as Apple's finished the current batch of major header-file changes. Source-code-level debuggers are coming. Levels of performance and abstraction are rising. The competition's hot. I was impressed by the commitment that these companies are bringing to the programming tools market.

And the programmers! *Ai carambacita!* I wish a few Martian anthropologists could've followed the crew of Macahologists who snaked through the San Francisco streets toward a Szechuan Chinese foodery Saturday night for what's become a main Macworld Expo happening—the Netters' Dinner. Combustible comestibility for a group of cybernauts who usually meet at long electronic distances. Hard to tell which was hotter/faster: the food, its consumption, or the idea flow. What is the link between this kind of cuisine and the programming mind, anyways?

Code Corner

Alright, chitchat's over. Let me calm down by discussing some highlights of the HyperCard Scouting Toolkit (ST) project whose images and code sources were shown in April. Please refer to that column for figures, listings, and an introduction to the project.

A Quick Review

Stacks are composed of one or more cards. Backgrounds hold features that are common to one or more cards. Cards can contain buttons and text fields and graphic designs. Backgrounds can also contain buttons and text fields and graphic designs. Several stacks may be linked by buttons so that they form a set of stacks.

HyperTalk programs consist of a series of message handlers. Things happen when you work with HyperCard, and those events cause messages to be sent to various HyperCard objects: buttons, fields, cards, backgrounds, stacks, external code resources, and the HyperCard program itself. A message passes up a hierarchy leading from simple objects to increasingly powerful ones, until one of those objects deals with the message by handling it and not passing it on. An object's message handlers are contained in its script. Take a look at the various HyperCard books, documentation sets, and help stacks for diagrams of that hierarchy and more detailed descriptions of the flow of control.

If you use the Scouting Toolkit for a while, you'll want to tweak it.

The Stack, the Background, and the Card

The Scouting Toolkit lives on a stack that has one background and one card. The stack has no script. Neither does the background. And the background contains neither fields nor buttons nor graphics. Backgrounds are useful in stacks with more than one card, where they are used to hold information common to several cards. In this simple stack, there's no need for the background to do anything.

The ST card, which I'll call the STBC (Bat Cave), contains 17 buttons and 2 fields. The STBC has a script

with two message handlers. Refer to last column's Figure 5 for a picture of the card and last column's Listing One for the script (as well as the scripts of all the other objects in the stack and brief descriptions).

When the card is first opened up, it's sent an *openCard* message. I'm nutty about Zen-like workspaces, so this card's *openCard* handler hides several things: the menu bar, the message box, the tool window, and the pattern window. Later on, buttons on the toolkit let you toggle the visibility of those items. Minor problem in toggling: There are HyperCard functions that tell whether the last three are visible, but no such function exists for the menu bar. So the *openCard* handler stores an indicator in a global variable—one that's known throughout the stack world—imaginatively named *menubarState*.

If someone plays around with the objects on the STBC, certain buttons might go into hiding. A click anywhere in the STBC outside a button will be caught by the card's *mouseUp* message handler. It uses a simple loop to make all the card's buttons show up.

The Buttons

Button 1 is one of the most complex. It's the button that kills the Scouting Toolkit, deleting all its buttons and fields from a card, including itself. Button 1's script handles one kind of message, a *mouseUp*. The button will not delete the ST from the STBC; in that case, it'll flash a warning field at the user. On any other card, though, the message handler removes each button and field via a Victorian clockwork sort of process, as follows.

It turns the cursor into the button tool. For each button it then puts a click at that button's location to select it and calls the HyperCard Edit menu's Clear Button command. Next, it turns the cursor into the field tool. For each field it then puts a click at that field's location and calls on the Edit menu's Clear Field command.

These are two nice HyperCard features: simulating mouse actions and making menu selections. Note how the handler can remove the button whose script it's in. Also, note how I make sure the *userLevel* variable's high enough to let the handler pull off these tricks and am careful to restore the original *userLevel* when done. (*UserLevel* is a global-state variable that controls access to various levels of HyperCard features.)

Button 14's also interesting. It's the button that replicates the Scouting Toolkit onto a new card. On a mouse click, it copies itself to the Clipboard. Then when you paste it onto a new card, it brings over the entire ST and kills itself. How? Well, once pasted on the replication target, it receives a *newButton* message. The *newButton* handler starts a series of trips, going back to the STBC, copying a button, returning to the target card, and pasting the copied button in. Once all buttons have been transferred, it gets rid of itself, using the technique described earlier.

This process is slow. Using HyperCard's Lockscreen command, so HyperCard doesn't have to draw the screen on each stack round trip, speeds it up somewhat. Still, it'd sure be easier if multiple stacks could be open simultaneously.

Button 3, used to open up the ST, passes a button click on to another button. Button 4 plays music to accompany and emphasize its action. Buttons 5–8 are simple toggle switches. Buttons 9–12 use menu commands. Button 13's analogous to button 2. Button 15 passes a button click on to button 4.

The Fields

There are only two fields. Field 1, normally hidden, shows up and displays a copyright notice under the control of button 15. Field 2, also normally hidden, shows up when someone tries to click button 1 from the STBC. Hidden fields are a neat way to add a sense of animated intelligence to your HyperCard work.

Going Further

If you use the ST for a while, you'll want to tweak it. Two examples that I've worked on are installation into a background and being able to change the spatial configuration of the buttons, with the ST remembering that change. What's interesting is that low levels of object self-manipulation can be done in straight HyperTalk, without resorting to XCMD and XFCN work.

The more I work with HyperTalk/HyperCard, the more I appreciate the convenience, ease, tweakability, and power of the tool. There's a certain simple pleasure to working in its environment, one I haven't had since early BASIC on early micros.

Now, if they can just reduce isolation and modality and get that darned execution speed up. . . .

Reader Mail Snapshots

Thanks to all of you who've proffered feedback—oral, electronic, paper—on my first column (January). I appreciate each piece, pro and/or con. It's a major navigational aid. A few quick gleanings:

Wayne Pollock mentioned the gap in Mac literature between books for novices and those for experts and expressed hope that this column might fill part of that hole. I agree,

TO THE MACS
(continued from page 86)

and I'll try. And Scott Knaster's new book, mentioned earlier, should be helpful. Wayne also wants more skeleton code and examples. Good news on that front from Apple itself, where the Developer Technical Support team has started work on a major new code examples effort. And this column will sport a small Multifinder skeleton Real Soon Now, as Prof. Pournelle would say. Wayne also wants some C++ material. Well, as mentioned earlier, I've finally begun fiddling with MPW, partially because I sniff that a Mac C++ will show up in that environment first. When it does, I'll muck about and report back.

James Savidge wrote requesting a little more information about the solitary useless Mac programming tool I mentioned in my first column. It was a C compiler, not one you see advertised anymore, but beyond that I say nothing. I refuse to give a company that won't refund money to a completely dissatisfied customer any form of energy. Besides, I haven't heard or seen squat about the hucksters for years, so hopefully their tent's collapsed and they've sunk back into the dark.

Several programmer buddies were happy to see more Mac coverage here in *DDJ*. I agree. But my favorite piece of mail came from a guy whose name I don't know—the missive's enshrined somewhere down at *DDJ* worldwide headquarters—who was P.O.'d at all the graphics that accompanied the January article. He reads *DDJ* for code, not cartoons. Now *that's* the *DDJ* audience I love! Keep it coming.

Updates and Fixes

Apple's excellent *Human Interface Guidelines* book that I mentioned in the March column is now out in trade form from Addison-Wesley. A lot of the Apple docs will make it out via that route: drafts through APDA, finished stuff via A-W. It's called the *Apple Technical Library*. The folks from Reading do a nice production and finish job.

SuperMac Technology's stopped manufacturing the Enhance board I mentioned in the January column. Too bad, it was/is a fine product. But we've got the Levco stuff now. Also, the final fix for the parasitic clip problem I mentioned regarding my Enhance board: a piece of carefully connived corrugated cardboard artfully wedged twixt the daughterboard and the clip, applying that constant pressure so needed for reliable electromechanical connection—Kludge Klassic #456.

Minor (Ha! Aren't they all when you own 'em?) bug in the March column's source code: the second instruction in the *doDrawCntl* routine, shown on page 70, should branch to that routine's closing *RTS*, not to the label drawn. So put a label on the *RTS*, say *DoDrCnBye*, and change the line to:

```
BEQ doDrCnBye ; it's invisible, so
no need to draw it
```

This is a branch rarely taken, and the flawed version has a solid chance of not flaming a program. Classic formula for insect survival.

Shut Up and Wrap-Up

Yow, it's time to recede. Thanks to all the nice Maccites at the Expo who took time to share thoughts and endure my nonstandard interviewing techniques, including (but not limited to, as I'm prone to quirky memory and misplaced interview tapes) Harvey Alcabes, Scott Boyd, Frank Catalano, John Draper, Joseph Edozien, Chris Espinoza, Amy Goldsmith, Michael Gosney, Michael Green, Ray Heizer, Glenn Hoffman, David Intersimone, Steve Jasik, Margie Kaptanoglu, Bill Kelly, Scott Kim, Scott Knaster, Richard Koch, David Krathwohl, Lance Lewis, Ken Loomis, Greg Marriot, Julia Menapace, Jeff Nutt, Peter Olson, Howard Pearlmutter, J. Scott Phillips, Heidi Roizen, Gerard Schutten, David E. Smith, Joel Spiegel, Michael Swaine, Wes Thomas, Randall Tinkerman, Neal Trautman, and Steve Splonskowski. With special thanks to Amos Gottlieb and roommates for the fine and classic Haight Street gestalt accommodations.

Next time out? I've given up the prediction racket. My batting average is invisible. But, hey, it'll be fun no matter what, eh? Software! Books! People! Code! Especially code, because I've taken it light this month. So dive happily into those fine mind exercisers y'all are so addicted to, and come back ready to do the logic boogie.

Bibliography

Apple Computer Inc. *HyperCard Developer's Toolkit*, Version 1.0. Available through APDA. Part #KMS036.

Apple Computer Inc. *HyperCard Script Language Guide*. Available through APDA. Part #KMB009.

Knaster, Scott. *Macintosh Programming Secrets*. Reading, Mass.: Addison-Wesley, 1987. ISBN 0-201-06661-0.

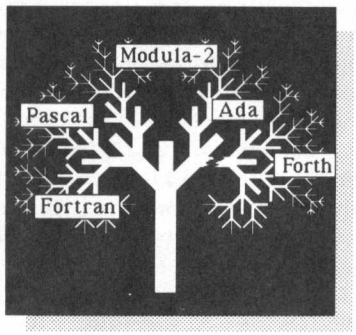
DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Vendor

APDA
290 S.W. 43rd St.
Renton, WA 98055
206-251-6048

Convincing Pascal to Read Non-Pascal Files



Sometimes a feature of a language is merely a defect put in a favorable light. It all depends on what you're trying to accomplish. Pascal, for example, insists that all files be bound to a data or record type: very noble from the standpoint of preserving the purity of strong typing but often an obstacle when trying to process files formatted in languages other than Pascal.

Specifically, the kinds of files I'm talking about are self-describing tables such as those generated by dBASE, Reflex, and other database programs. Typically, such files begin with several data structures describing the contents, followed by any number of fixed-format data records. It's easy to mix record types with C and assembly language and even BASIC, all of which support free-form files. You have to convince Pascal to do it, though, and the trickery for doing so is the subject of this month's column.

I'll also respond to a reader's complaints about Turbo Pascal 4.0.

Creating a Table File

Rather than covering a specific vendor's table format, I've developed a simple model for this article that is typical of these files in general. What makes it simple is not the file structure itself but instead the number of options. Data records can consist of only two field types: integers and character arrays. The thrust here is the principles, and there's no sense muddying the waters with a number of options that you can figure out for yourself.

by Kent Porter

The typical table file begins with a fixed-length preamble (256 bytes in this case) containing a header record and field descriptors. The header record is a fixed structure that contains several fields giving basic information about the contents. In this case, the header record is 24 bytes long and contains the fields shown in Table 1, page 95.

Signature is an invariant value written at a fixed place to identify the file as belonging to the application. If some other value appears in that position, the file doesn't follow the rules given here and can't be processed. The value of *signature* for this application is 19364 (4BA4h), and it appears in the first 2 bytes of the file.

The *nrecs* field tells you how many data records the file contains. *Tablename* is a packed array of ten characters giving the name of the table; not all vendors have an analogous field in the header record. *Datastart* expresses an offset, with respect to the start of the file, to the first data record. This is a *long* (32 bits on a PC) integer to correspond with the usage of *fseek/ftell* in C. Turbo Pascal 4.0 and Microsoft Pascal similarly use a *long* integer for their *SEEK* procedures. The last two fields, *descrsize* and *ndescr*, have to do with the next part of the file preamble.

A data record consists of one or more fields, each of which has three attributes: a name, a data type, and a length. These constitute a field descriptor, which has the form shown in Table 2, page 95. Each data field has one descriptor, hence there are *ndescr* field descriptors of *descrsize* following the header record. In the programs that accompany this article, for example, there are two fields (*ndescr* = 2), so there are two descriptors.

Thus the preamble consists of a fixed header record followed by a variable number of descriptor records, each of which has a fixed format. Taken together, they describe the data content of the file. The unused portion of the preamble is filled with uninitialized garbage.

The data itself begins at byte offset *header.datastart* from the beginning of the file. Each record corresponds to a row in the data table and each field to a column. The descriptor records describe the columns, and the length of any given record is the sum of all *flen* fields in the descriptors. There are *header.nrecs* records. The file is thus a self-describing entity, and the program's job is to interpret the descriptions in order to extract the data. Figure 1, page 95, shows the format of a simple table.

Listing One, page 69, is a generic C program (MKTABLE.C) that creates a table with the preamble described here. The program then requests data entry and writes out the data records you type in response, saving them in a file called database.xyz. In order to build an adequate table, enter at least three or four records. Terminate data entry by pressing Return when the program asks for a name. The program then updates the header record to reflect the number of data records entered and closes the file. This is a vastly simplified version of a database management package, but it generates a complete table of the same sort that flows out of dBASE and other table-oriented database products.

Translating Strings

The references to *pac* (a packed array of characters) in Figure 1 and Table 1 point up a fundamental difference between Pascal and lower-level languages such as C and assembly language. Although not de-

fined in the academic standard, the *pac* string type is supported by most real-world Pascal compilers. It's really a stretched *PAC*, the difference being that element 0 contains the string length, with the first valid character at element 1. Thus, at the data level, a string containing the word *Pascal* looks something like *6Pascal*. If the string were declared as:

```
VAR strng : STRING [10];
```

the last 4 bytes would contain garbage. The compiler inserts string-handling routines that pay attention to the length byte.

C handles string data differently, and most assembly-language programmers use the same convention as C. It's so common, in fact, that it has a name: ASCIIZ. In ASCIIZ, the 0th element contains the first character of the string. There is no length indicator; instead, end-of-string is signified by ASCII value 0, or *CHR (0)* in Pascal notation. This is merely a packed array of characters with a special end sentinel. The C term for it is a *null-terminated* string.

The *asciiz* function in Listing Two, page 70, transforms ASCIIZ strings into Pascal strings. Because it's possible that a string might exceed its maximum length, the function also takes the *max* parameter, which limits the number of characters in the result; it's either *max* characters or everything up to the null terminator, whichever comes first.

This string-translation routine and the liberal use of variant records are two keys to convincing Pascal to read non-Pascal tables. The third is processing the file on a byte-by-byte basis. Here's how it works.

Processing the File

The program declares the table as *FILE OF BYTE* and opens it. The first step reads the 24-byte header record into the stream variant of the *headrec* structured variable. This is necessary because the file is of type *BYTE*; all file reads are done in the same way. Access to the data elements is via the other variants, as in the next step, which checks *signature*. Execution continues if the *signature* is correct and ends with a message otherwise.

Procedure *showHeaderInfo* lists information from the header record. Note that the table name is drawn from the second variant of the *headrec* record. Why? Because the *asciiz* function expects an argument of type *pac*, which is 20 bytes long, whereas the real *tablename* field is only 10 characters long. This is a trick to prevent the compiler from choking on a mismatched type.

The *getDescriptors* procedure, called by *showHeaderInfo*, reads the field descriptors that follow the header file. The program assumes a maximum of ten fields for the table when it declares the *field* variable, which is an array of *fieldrec* structures. The *header.ndescr* variable governs the actual number read from the file. *ShowHeaderInfo* uses the descriptors to display information about the fields. The *showData* procedure uses them more extensively.

Before calling *showData*, however, the program first calls the Pascal *SEEK* procedure. The purpose is to reposition the file pointer to the start of the data records, which is past the unused portion of the preamble. *ShowData* can now read and process the table's data contents sequentially.

A couple of local variant record types provide the means for fetching integers and ASCIIZ strings. Again, the stream component furnishes type compatibility with the file. Because the host processor and not the compiler establishes the format of the integer type, an integer taken from the file as two consecutive bytes can be plucked directly from the variant without translation. The character field is accessible via a call to *asciiz*.

A pair of nested loops control the reading and display of data fields. The outer loop repeats for the number of records in the file, as given by *header.nrec*. The inner loop processes individual fields, stepping through the array of descriptors in order to determine what to read next from the file. Because it's loop-driven, *showData* can process any

Name	Type
signature	word
nrecs	word
tablename	pac [10]
datastart	longint
descrsize	integer
ndescr	integer

Table 1: *Names and types of fields in the header record*

Name	Type
fname	pac [20]
ftype	integer
flen	integer

Table 2: *A data record's field descriptors*

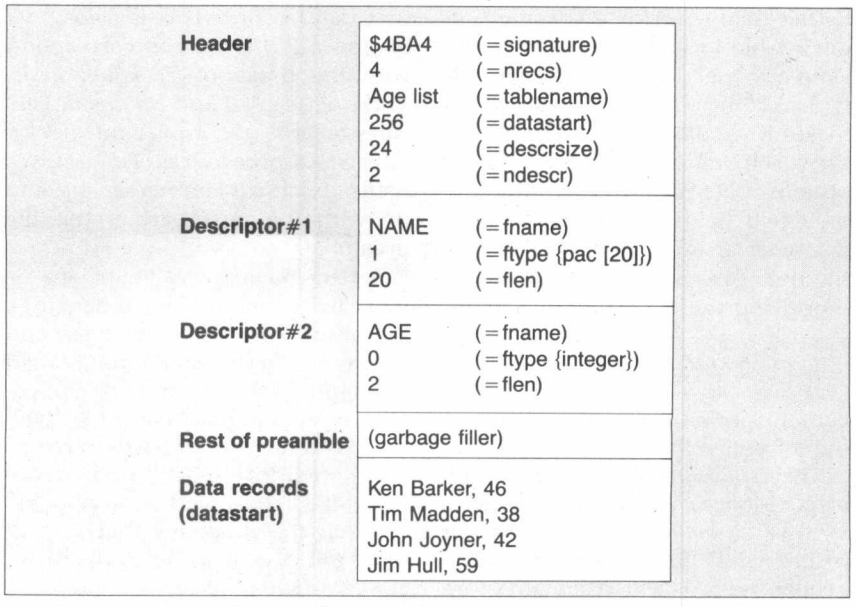


Figure 1: *Format of a simple table*

number of records consisting of any number of integer and ASCIIZ fields in any order without modification. Additional data types would require the appropriate structure definitions and expansion of the *CASE* statement.

This is not a complete table system, of course, because it lacks date, floating-point, and Boolean types. Also, it's not compatible with any existing database package's file format. Given the specifications for a file and the techniques presented here, however, you should be able to write a Pascal program that reads non-Pascal files with header records.

Turbo Pascal 4.0 Flames

The mail the other day brought a letter from Charles Linett, who heads up the Computer Science Staff at the Census Bureau. Charles and his folks use Turbo Pascal for communications programs of 15,000+ lines, and he's not amused by Version 4.0. Here's part of what he has to say:

"I see two rather large defects bordering on the semicalamitous for our type of work.

1. There are no overlays (as there were in Version 3.0). Borland has solved that problem in two suave ways, however. First, the company told us that if we used overlays, then we needed only to rewrite our programs (thanks, fellas, you're a big help). Second, Borland found the part of the documentation least likely to be read (a file called Q&A) and wrote that it recognized the need and was working on something 'intelligent.' I can only hope it gets it done before those awaiting the feature die of old age.

2. The manual is awful and (worse yet) has almost no chance of improvement. It requires so many additions and corrections that what is called for is a new manual altogether. If 20 pages need changing in a real manual, you send the customer those 20 pages and let him or her stick them in the book. This cannot be done with the 4.0 manual because it's glued together in one big lump."

No quarrel with the first point, Charles; I'll get back to it in a minute. As for the second, probably a lot of us aren't crazy about a bound book as a manual. And that one in particular is too thick; you can't spread it out on the desk for reference without either breaking the spine or putting barbells on it for paperweights. But the adjective *awful* is kinda harsh. Versions 2.0 and 3.0 had bound manuals, too, and Borland isn't the only company whose docs come this way.

Nobody's ever told me this, but I suspect the purpose of bound docs is to discourage pirates from photocopying the manual. Maybe if the world was a more honest place, vendors such as Borland wouldn't resort to defensive tactics. Piracy is just another name for theft.

Yeah, I don't like the manual either, but it's a whole bunch better than its predecessor in terms of both quality and content. And manual corrections conveyed via

READ.ME files are hardly a Philippe Kahn innovation.

Now for the overlay fiasco. No doubt about it, Borland shot itself in the foot by dropping overlays. Probably it figured it could get away with it because Version 4.0's .EXE files break through the infamous 64K barrier of Version 3.0's .COM files. Somebody should have surveyed the user community before Borland yanked the rug from under it.

But there's an alternative for Charles and anybody else who got abandoned. It's a product called Overlay Manager 4.0 from TurboPower Software (3109 Scotts Valley Dr., Ste. 122, Scotts Valley, CA 95066; 408-438-8608). Costing $45, this is an interactive program that lets you break a compiled .EXE file of any size (up to about 1 Mbyte) into any number of overlays. For truly enormous programs, there's another utility in the package that effects chaining. The slim 30-page manual is excellent and so's the quality of the software; TurboPower produces good stuff. Highly recommended if you need overlays.

Availability

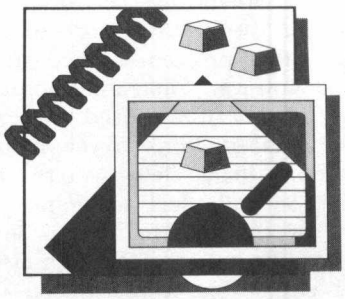
All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 69.)

Vote for your favorite feature/article.
Circle Reader Service No. 7.

Parallel Processing, Object-Oriented Programming, and a Reading List



Several of the key difficulties of my original text cluster about the concept of a paradigm. . .One sympathetic reader. . .prepared a partial analytic index and concluded that the term is used in at least twenty-two different ways.

—Thomas Kuhn

Authors who use the word *paradigm* seem to permit themselves to mean by it a large number of different, and perhaps contradictory, things; and as I am committing myself to dealing with paradigms each month, I will honor this wise and liberal tradition.

A column called Programming Paradigms should, though, at least, deal with the issues Robert Floyd brought up in his 1978 Turing Award Lecture, "The Paradigms of Programming" (even though Floyd didn't define paradigm either). Floyd deplored the segmentation of computer science into narrow communities, "each speaking its own language and using its own paradigms. . .well-defined schools of LISP programming, APL programming, ALGOL programming, and so on." Such communities develop not just around languages but around any broadly applicable problem-solving technique, including such diverse techniques as recursion, stepwise refinement, generate-and-test, and data-flow design.

Programmers advance in their careers by learning a community's shared techniques, terminology, values, prejudices, model problems,

by Michael Swaine

and concrete examples of how to solve such problems—what Thomas Kuhn, who investigated scientific paradigms, calls puzzle solving. This is the process of adopting a paradigm, and it prepares the programmer to answer an ad for a "C programmer" or a "software engineer" or a "knowledge engineer" or a programmer "with experience in object-oriented design."

A column called Programming Paradigms should offer some insight to the atypical programmer for whom this is not enough, the programmer who wants to add to his or her repertoire of paradigms.

This column will attempt to do that by exploring techniques from parallel processing, programming in logic, functional programming, object-oriented programming, and other alternative models. The focus will be on pointing out the alternatives to familiar, conventional methods. I can't think of a better place to begin the exploration than in the wilderness of MIMD parallel processing.

Parallel Paradigms

When I conducted an informal survey through the pages of the magazine last year, parallel processing came out at the top of the list of topics readers would like to see *DDJ* cover more often. It's not surprising; parallel processing promises increased throughput independent of other speedup techniques, and parallel processing raises new problems to be explored—a lot of them—the parallel world is bigger than the sequential world. There are many paradigms of parallelism, some better mapped than others. One of the least explored is MIMD parallelism.

The term *MIMD* means multiple instruction, multiple data. Logically, there are four such terms, representing the application of parallelism to instructions, data, neither, or both. The terms can be applied to both architectures and algorithms. SISD (single instruction, single data) is the familiar sequential Von Neumann model for computer architecture and programming; and MISD, in which multiple instructions are applied to single data items, turns out to be in practice indistinguishable from SISD. That leaves two broad paradigms of parallel processing: single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD).

SIMD breaks down into two lower-level paradigms, each with its community of practitioners, traditions, model problems, and typical solutions. These are array processing and vector processing.

The array-processing paradigm (see Figure 1, page 102) involves a sequence of instructions applied concurrently to disjoint sets of data. The ICL Distributed Array Processor, which consists of a control unit and a 64×64 array of processors, each with its own local memory, is one example of this paradigm. In it, the control unit drives the parallel processors, which all perform the same operation in lockstep on the data in their local memory. This paradigm is natural for matrix mathematics, graphics processing, and other uniform operations on arrays of data—hence the name array processing.

The vector-processing paradigm, which is also called pipelining, involves instructions overlapped on disjoint sets of data (see Figure 2, page 103). With pipelining, additional operands can be fed into an operation before it has finished with the first because the operations are broken down into stages and the stages are processed in parallel.

You can see architecture designed for pipelining in supercomputers such as the Cray-1, which are often called vector processors because they process a vector of operands in

(continued from page 100)

parallel. You can also see pipeline architecture in the microprocessors of desktop computers. The Motorola 68030, for example, can fetch both instructions and data from on-chip caches, refresh the caches from off-chip memory, and ready the address for an off-chip fetch, all in parallel. Pipelining in CPU architecture and operating system design is a well-established technique. Extending this technique to a computer system as a whole presents a new paradigm, presently seen chiefly in supercomputers.

Both vector processing and array processing are single-instruction, multiple-data parallelism, and consequently they exhibit the traits of SIMD. SIMD approaches typically involve a high degree of parallelism; that is, many array or vector elements are processed at once. They restrict this parallelism to a low level, such as the level of the operation. They exhibit central control and strong synchronization. As a result of these traits, SIMD approaches concentrate most of the need for special algorithms at the system level and present no special problems in communication among the parallel elements.

MIMD Parallelism

In contrast, MIMD approaches usually involve fewer but more powerful processing elements, with a medium degree of parallelism and parallelism at a higher level—the level of the task. Control is distributed, and synchronization is only occasional. As a result, MIMD paradigms raise difficult problems in communication among the parallel tasks as well as higher-level problems in the design of algorithms.

In an MIMD architecture, different parts of an application program are given to different, independent, networked processors, each with its own instruction set, each capable of performing a sequence of instructions without supervision.

At the algorithmic level, MIMD parallelism means decomposing the problem into components that can be attacked in parallel via separate processes. This is a true paradigm shift; you envision the problem differently in MIMD parallelism from the way you envision it in SIMD parallelism or in a sequential paradigm. SIMD is a paradigm shift away from SISD, but MIMD is a shift away from SIMD and it's a bigger shift.

If Henry Ford's assembly line is

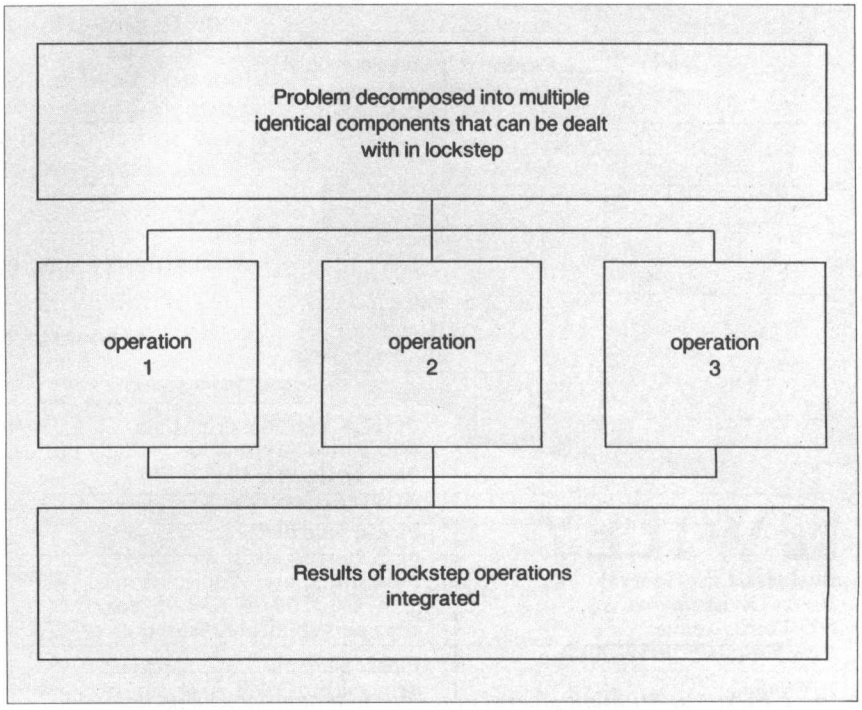


Figure 1: *Array processing applies the same operations to many data sets at the same time, achieving a lockstep parallelism.*

the metaphor for vector processing, then the array-processing metaphor is all those WACs plugging away at elementary arithmetic operations in lockstep back during World War II. MIMD parallelism is closer to the model of research within a scientific community, where, in pursuit of a common goal, individual researchers handle their own tasks, sharing information when it seems mutually beneficial—ARPAnet, including the networkers.

MIMD architectures (see Figure 3, below) are to date mostly experimental. Shared memory and nonshared memory architectures have been tried, as have been various topologies for the linking of the processors. The Denelcor Heterogeneous Array Processor and Cray X-MP are two commercial MIMD machines, and CalTech's Hypercube is a model that has been generating a lot of interest this year. But MIMD architectures are moving onto the desktop, and it is already possible to experiment with MIMD parallelism for the price of a fully loaded personal computer.

Foreseeing this possibility, Les Record, of Round Rock, Texas, sent me the following suggestion:

"Your discussion of problems in parallel computing brought to mind an idea for a major *DDJ* project: using minimal hardware, say 6 to 8 Z80s, interconnected in a simple network, present a series of articles using parallel techniques to solve simple (?!?) problems. If you don't want to present a hardware article (one every 2–3 years doesn't hurt), then perhaps some OEM might contract to build a minimal parallel system. Does this sound feasible? I think the hardware could be low-cost. The pay-

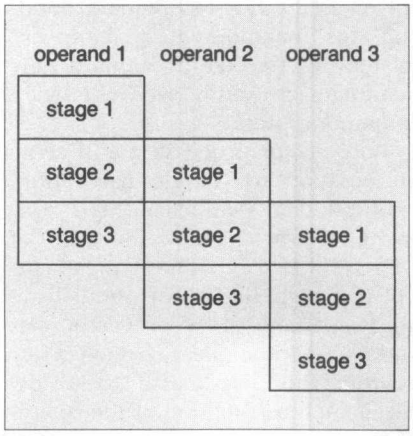


Figure 2: *Vector processing, or pipelining, achieves parallelism by overlapping the components of a multicomponent process, much as automobiles are built in parallel on a Detroit assembly line.*

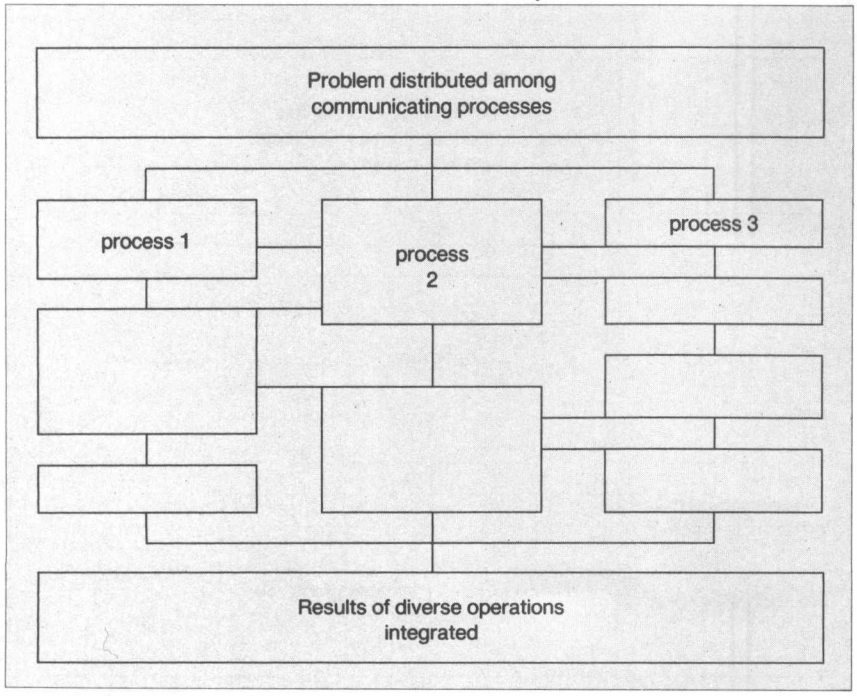


Figure 3: *MIMD (multiple instruction, multiple data) parallelism is typically asynchronous, with independent processes communicating with one another to achieve occasional synchronization.*

Sources

A column such as this can't begin to deal with its subject matter in the depth that, say, a column on C programming for MS-DOS can. I have only touched on many important issues in this installment. So here are some places where you can find the issues treated in greater depth.

You can find a good overview of parallel programming techniques and architectures in *Parallel Programming* by R.H. Perrott (Addison-Wesley, 1987). David Harel's excellent *Algorithmics: The Spirit of Computing* (Addison-Wesley, 1987) has a good section on some of the issues in parallel processing.

I also found several books from Springer-Verlag very useful, including *WOPPLOT 86, Parallel Processing: Logic, Organization, and Technology* (Springer-Verlag, 1987) and *PARLE, Parallel Architectures and Languages Europe* (Springer-Verlag, 1987), which report on conferences held, respectively, in Neubiberg, Federal Republic of Germany, in 1986 and in Eindhoven, the Netherlands in 1987.

The best source of information about the INMOS transputer is INMOS itself, through various technical notes. The U.S. contact is INMOS Corp., P.O. Box 16000, Colorado Springs, CO 80935; 303-630-4000.

Levco has video training and a developer group for learning about parallel processing, the transputer, and Levco's Macintosh boards. Levco is located at 6160 Lusk Blvd., Ste. C-100, San Diego, CA 92121; 619-457-2011. Levco's boards are not the only transputer implementation you might want to investigate; Definicon is another company that is doing interesting things with the devices. Definicon is located at 1100 Business Center Circle, Newbury Park, CA 91320; 805-499-0652.

Dick Pountain and David May give a very readable introduction to the transputer's high-level machine language, occam, in their *A Tutorial Introduction to Occam Programming* (INMOS/BSP Professional Books, 1987). Occam is still nearly undiscovered; this was one of only four books devoted to occam that I was able to find in a search of the entire University of California library system. Perrott's book also gives a clear introduction to Hoare's CSP and to occam as an implementation of CSP's central concepts as well as showing how parallel processing is implemented in a number of other languages, including Ada and Pascal Plus.

For explicit definitions of object-oriented programming, I drew upon "What Is Object-Oriented Programming," an invited lecture by Bjarne Stroustrup reprinted in *ECOOP '87: European Conference on Object-Oriented Programming* (Springer-Verlag, 1987); *Object-Oriented Programming: An Evolutionary Approach* by Brad Cox (Addison-Wesley, 1986); and the August 1981 and August 1986 issues of *Byte*.

But object-oriented programming is better defined by example in such sources as *Smalltalk-80, the Language and its Implementation* by Adele Goldberg and David Robson (Addison-Wesley, 1983); *The C++ Programming Language* by Bjarne Stroustrup (Addison-Wesley, 1986); and the documentation for Digitalk's Smalltalk-V and The Whitewater Group's Actor. Digitalk Inc. is located at 9841 Airport Blvd., Los Angeles, CA 90045; and The Whitewater Group is at Technology Innovation Center, 906 University Place, Evanston, IL 60201.

Among the sources of continuing education for object-oriented programmers are OOPSLA and the new *Journal of Object-Oriented Programming*, which you can get from SIGS Publications, 310 Madison Ave., Ste. 503, New York, NY 10017; 212- 972-7055. OOPSLA, the main conference on object-oriented programming, takes place this year in San Diego, Calif., September 25–29. The contact person is Barbara Noparstak, Digitalk Inc.; 213-645-1083.

This column had nothing to say this month about artificial-intelligence paradigms, such as functional programming, logic programming, and neural nets. The least I can do is to mention some good sources in these areas.

I am looking forward to my first issue of a journal devoted to LISP, called *LISP and Symbolic Computation: An International Journal*, edited by Richard Gabriel and Guy Steele. You can order it from Jan Zubkoff, *LASC*, Lucid Inc., 707 Laurel St., Menlo Park, CA 94025; 415-329-8400. Steele and Gabriel are, among other things, the authors of two important books on LISP—*Common LISP: The Language* by Guy L. Steele, Jr. (Digital Press, 1984) and *Performance and Evaluation of LISP Systems* by Richard P. Gabriel (MIT Press, 1985).

The Prolog language gets its name from the exaggerated notion that you are programming in logic when you use it. A new book removes some of the exaggeration by showing how to use and extend Prolog to do true logic programming; it's *Computing with Logic: Logic Programming with PROLOG* by David Maier and David S. Warren (Benjamin/Cummings, 1988).

A journal on neural network technology is available from *(TK)*. A conference on neural nets will also be held in San Diego, Calif., from July 24-27. It's sponsored by the IEEE and you can get more information from Nomi Feldman at 619-453-6222.

The big conference for the artificial-intelligence community is AAAI, held this year in St. Paul, Minn., from August 22-26. Call 415-328-3123.

Philosopher of science Thomas Kuhn made the term paradigm mean so much in his landmark book *The Structure of Scientific Revolutions* (University of Chicago Press, 1962). The disclaimer quoted at the beginning of this column is from the postscript in the second edition of this book, published in 1970. Robert Floyd's Turing Award lecture appears in *ACM Turing Award Lectures, The First Twenty Years: 1966–1985* (ACM Press, 1987)—a wonderful book. Most of the lectures in it not only deserve reading but also reward rereading. —M.S.

off would be exposure to software—hands-on! I might even be able to help."

I especially like that last sentence. I'm passing Les' suggestion along to Tyler and the gang. But if the goal is exposure to the programming problems of MIMD parallelism rather than the architectural issues, wouldn't it make sense to buy the hardware off the rack? This is now possible with the INMOS transputer.

Transputers

Interest in the transputer is growing. Last summer, the Macintosh SIG of the Software Entrepreneur's Forum polled its members for topics for future meetings, and the hands-down winner was the INMOS transputer.

The transputer is a 32-bit RISC chip designed by INMOS Ltd. for parallel processing. It includes a processor, local memory, and four dedicated I/O ports. Several transputers in a simple network can be used to implement MIMD parallelism. The transputer can switch between parallel tasks in a microsecond.

Transputers can be linked in a network, with a great deal of flexibility in the topology of the network and in the physical location of the nodes: transputers on the same network need not occupy the same circuit board, the same system bus,

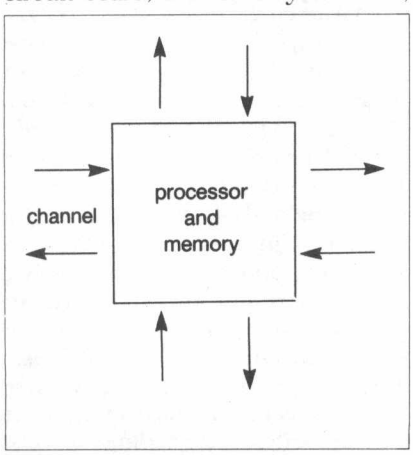


Figure 4: *One transputer processor. Independent processes can run in parallel on individual transputers, communicating with one another via the four channels each transputer has.*

or the same city. Intertransputer links are point to point, so the size of the network is not limited by contention problems as it would be on a common bus. And the number of transputers in the network can be increased or decreased without altering the parallel program running on the network. (See Figure 4, page 105.)

The transputer is already being used in commercial products. Two companies have announced transputer-based boards that will significantly speed up laser printers.

The domain of MIMD algorithms is still wide open.

Both CSS Laboratories and Eidolon will be showing laser printer controller boards this year, each employing one T414 transputer as a coprocessor for throughput in the 8–20 page-per-minute range. Eidolon has also announced a two-transputer controller that uses parallelism to get 40 ppm throughput.

HRC Micro Organization Ltd. has a CAD product called Intervision that uses transputers to achieve a 10 to 40 times speed improvement. Atari demonstrated a prototype of a transputer-based product, the Abaq computer, last November (at Comdex, a terrible place to introduce genuinely new ideas). The Abaq will use T800 transputers (it has space for a dozen) and run Helios, a Unix-like operating system, with an MS-DOS emulation mode that should run DOS programs faster than an AT. Penguin Software is developing Unix-based transputer development tools. And Levco is selling transputer boards for personal computers.

It's Levco's boards I had in mind when I alluded to building a parallel-processing system from off-the-rack parts. Levco allows you to turn a Mac II or SE into a parallel-processing system with a package it calls TransLink. TransLink consists of a bus card, transputer modules, and either MPW-compatible development software (C compiler, transputer assembler, loader, linker) or the occam development system from INMOS. The transputer modules include 256K to 4 Mbytes of RAM in four SIMM sockets and one transputer (T414 15 MHz, T414 20 MHz, or T800 20 MHz with 64-bit IEEE floating-point support). The bus card for the Mac II can hold up to four transputer modules, whereas the SE card can hold only two.

Fully loaded with five TransLink Nubus-compatible boards, holding a total of 20 20-MHz T-800 transputer modules, each with from 256K to 4 Mbyte of its own RAM, a Mac II would reach a throughput of nearly 200 MIPS. (If only I weren't already in debt up to my neck for the Mac II. . . .)

Coming down from the clouds, I should explain that Levco's boards provide an opportunity for developers to explore the largely unknown territory of MIMD algorithms.

MIMD Programming

The domain of MIMD algorithms is still wide open. There are many hard problems to be solved, including questions of appropriate topology, general techniques for functional decomposition, and specific MIMD algorithms for familiar problems. Small-scale functional decomposition, with just a few processes running in parallel, can produce performance unattainable in any other way.

It is only fair to point out, though, that there are those who doubt the benefits of large-scale MIMD parallelism. Their arguments rest on a strict upper bound on those benefits: the limit set by Amdahl's law.

Amdahl's law states that parallel speedup (the ratio of the speed of processing using any parallel technique whatsoever to the speed of the strictly sequential approach) is bounded above by $100/[(100-f)+f/p]$,

where f is the percent of the total work that can be done in parallel and p is the number of processors. The value of f cannot be 100 because some communication and control overhead is necessarily not parallelizable. This means that, with five processors and an algorithm that is 75 percent parallelizable, the maximum speedup attainable is only 2.5. Adding more processors will only increase the speedup gradually toward a final limit (for this particular algorithm) of 4.0. Given that communication overhead can increase rapidly with the number of processors in an MIMD system, the value of additional processors drops quickly to nothing in this scenario.

Nevertheless, even large-scale MIMD parallelism is worth looking into. The problem with the scenario just discussed was the 75 percent parallelizability. As the value of f in Amdahl's equation approaches 100, the entire expression approaches linearity; that is, n processors yield an n-fold speedup. There are experimental results showing that in interesting cases near linearity is in fact attainable. For one example, see McBurney's comments in *PARLE, Parallel Architectures and Languages Europe* (Springer-Verlag, 1987).

Relative speedup, which is the speedup due to parallelization minus overhead, approaches linearity as the number of processes (that is, tasks, not processors) increases (but it has to increase a lot). Of course, it is exactly the nonparallelizable overhead that sets the limit in Amdahl's equation; but when, as is the case in a wide variety of problems discussed by McBurney, the overhead cost becomes negligible with increasing problem size, then even large-scale MIMD parallelism suddenly makes sense.

MIMD puts more burden on the application programmer to partition the program into appropriate components for parallel execution. Languages have been developed to facilitate this partitioning, some of them deriving from a model developed by C.A.R. Hoare called Communicating Sequential Processes, or CSP.

CSP

Hoare's model is of two or more independent processes, each consisting of a sequence of instructions executed sequentially. The processes are no different from programs written in any conventional sequential programming language, except when they must communicate with one another. In Hoare's model, this can only happen when each of two processes desires to communicate with the other process, and he calls this synchronization of desires a rendezvous.

Hoare's rendezvous is a strict synchronization, and by itself it would defeat one of the canons of parallelism, which is to keep all processes busy as much of the time as is possible. To free a process that might otherwise be locked up waiting for a rendezvous, Hoare adopted nondeterministic alternation and repetition constructs from Dijkstra.

These nondeterministic constructs require something called a guard, which is a Boolean expression that precedes a command. Only if the guard is true is the command executed, but its truth does not guarantee execution. The guarded alternation construct causes at most one of a set of guarded commands to be executed, with that one selected at random from those commands with true guards. The guarded repetitive construct causes all commands with true guards to be executed until no guards are true.

Guarded alternation and repetition allow a process, for example, to pass on data to any other available process or to get data for processing any time any other process has data to deliver.

Together, the concepts of sequential processes communicating via rendezvous and nondeterministic alternation and repetition make up a model for MIMD parallelism that is both provable and a solid basis for a programming language. One such language, and the natural one for developing transputer-based software, is occam.

Occam

"Occam's Razor slices things down to simplest causes. Single causes have a fair chance of being right."—from "Occam's Scalpel" by Theodore Sturgeon (*Worlds of IF*, August 1971).

Occam's razor (or sometimes Ockham's razor) is the principle of ontological economy, attributed to William of Ockham (or Occam), circa 1285–1349, an English Franciscan, heretic, and philosopher best known among philosophers for his antirealist interpretations of universals. To the rest of us, if he is known at all, it is for Occam's razor, which states that, in explaining nature, entities should not be multiplied beyond necessity: the simplest explanation is the best.

The programming language occam (with a lowercase *o*) was designed by people at INMOS for writing parallel programs to run on the INMOS transputer processors. Occam supports parallel processing with only a few new programming entities. So clean is the occam implementation that major portions of the occam development system have been formally proved correct.

Occam implements the key concepts of CSP directly by the use of constructors. Sequences of commands (also called primitive processes) are grouped into sequential processes using the constructor *seq*, concurrency among such processes is indicated by the constructor *par*, and nondeterminism is introduced with the constructor *alt*. Nondeterministic alternation and repetition are implemented using the *alt* constructor in conjunction with Pascal-like alternation and repetition constructs such as *if* and *while*. Communication between processes follows the CSP rendezvous model, with two processes choosing a common channel on which to signal or pass data.

Although occam is a high-level language, it is also in some sense the machine language of the transputer, which was designed using occam. The two are closely linked. In particular, occam processes are run on individual transputers, and occam channels map directly onto the physical transputer links.

In Example 1, page 115, I give a taste of occam code. The three processes INPROC, BUFFPROC, and OUTPROC run in parallel, with INPROC gathering data and passing it to BUFFPROC, which in turn passes it to OUTPROC for output. Communication between processes in CSP occurs at a rendezvous and there are two in this example: when BUFFPROC calls INPROC and vice

versa, and when BUFFPROC calls OUTPROC and vice versa. Note that indentation is not optional in occam.

I hope this taste of parallel paradigms has been useful and entertaining. Thinking through parallel approaches can be enlightening even if you never expect to work on a parallel system. Several programmers have pointed out the possibilities for learning new sequential approaches from studying parallel techniques. And it adds to your paradigmatic breadth.

WHOOPS, or What is Object-Oriented Programming?

One programming paradigm that has become extremely popular of late is object-oriented programming. *Object-oriented* is the current vogue word, displacing *structured* as synonymous with whatever is good and true and beautiful in programming. Smalltalk, Actor, Simula67, and C++ are described as object-oriented; there have been claims that Ada, LOOPS, and APL are object-oriented languages; and there is much talk of object-oriented design in Pascal, Modula-2, C, and Forth. Just what is the object-oriented paradigm? What features does a language need to have in order to be said to support the paradigm? And what does it mean to say that one is programming in the paradigm?

Bjarne Stroustrup has come up with a set of definitions that help to clarify the issue, at least if you accept them. I do, chiefly because they give a clear picture of object-oriented programming as a paradigm, indicating what kinds of puzzles the universe sets an object-oriented programmer.

Stroustrup takes pains to distinguish the object-oriented paradigm from the paradigms of data hiding and data abstraction. Data hiding, he says, boils down to using modules. You can achieve the effect of data hiding in C, but Modula-2 makes the module a fundamental language construct. Stroustrup says that Modula-2 supports data hiding but C only enables it.

Data abstraction according to Stroustrup means programming with user-defined types. Any programming language that provides the means for doing this supports the data abstraction paradigm—Ada and C++, for two examples.

One thing that the data-abstraction paradigm does not permit is expressing a distinction between the properties of a type and the properties of instances of the type. Object-oriented languages, Stroustrup says, are those that support expressing this distinction, such as Smalltalk. The mechanism for doing this is inheritance.

Object-oriented programming is, for Stroustrup, just programming using inheritance, and it is, roughly, a superset of the other paradigms. Smalltalk is certainly an object-oriented language, or rather an object-oriented environment. (Strictly speaking, no language can be object-oriented by itself; object-oriented programming requires support from a programming environment as well as support from a language.) I'm not sure where this leaves HyperTalk, which seems to have an inheritance structure but which does not allow creation of new types of objects. In any case, he breaks down the object-oriented paradigm further, as follows:

- decide which classes you want
- provide a full set of operations for each class
- make commonality explicit by using inheritance

The benefits of the object-oriented paradigm come from the exploitation of the commonality among types, and identifying commonality

```
INPROC::

     ...
     (* input X*)
BUFFPROC ! X

     ...

OUTPROC::

     ...

     BUFFPROC ? X
     (* output X *)
```

Example 1: *Communicating processes in CSP.*

PROGRAMMING PARADIGMS

in the problem is the chief task that the paradigm sets the programmer.

Stroustrup's definition of object-oriented programming is not the same as David Robson's in Robson's classic *Byte* article of August 1981, in which he presents the class/instance distinction (and consequently inheritance) as optional. But Robson seems to be defining a concept rather than a paradigm. It also does not match Brad Cox's view of the centrality of the software IC metaphor to object-oriented programming. And it is at variance with Geoffrey Pascoe's view, expressed in *Byte* in August 1986 that information hiding, data abstraction, dynamic binding, and inheritance are all defining features of object-oriented programming. But Stroustrup argues that information hiding and data abstraction are subsumed within inheritance.

One thing that Stroustrup's definition does provide is a picture of the kind of puzzles the object-oriented programmer faces as a result of being an object-oriented programmer. The object-oriented programmer selects a set of classes, provides operations for each class, and then sets out to identifying commonality in the problem in order to make it explicit by using inheritance.

It's explicitly a definition of object-oriented programming as a paradigm.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

EXAMINING ROOM

PC/Forms

Product:
PC/Forms, Version 1.21b
Target:
IBM PC, IBM PC AT, IBM PS/2, and compatibles
Requires:
DOS 2.0 or later; one floppy drive; 256K
Pricing:
C version $149.95; Turbo Pascal version $99.95
Vendor:
Golden Solution, P.O. Box 22216, Cleveland, OH 44122; 1 800-338-6754

Let's face it, screen layout and data validation are a drag. Most serious PC applications probably devote over half the code—and well over half the development effort—to tweaking displays and protecting the user from the GIGO syndrome.

Golden Solution's PC/Forms was designed to reduce this tedium and it does! In 15 minutes to half an hour, you can design a display, set up elaborate input validation criteria, test the form interactively, and generate the code to implement it. Writing and testing the equivalent source code from scratch could easily take several days.

The Turbo Pascal and C versions of PC/Forms are comprised of different sets of tools. We tested the C version, so that's what we'll talk about here.

The heart of the product is a stand-alone editor called FORMS. You run it to set up the display and validation and to generate the files used by your application. FORMS has bouncing-bar menus a la Lotus 1-2-3. It's a highly visual environment, with windows popping up all over the place and pick lists and such, all of which have an intuitive feel. In layout mode, you paint your form on the display using function keys and the usual editing commands. There are commands for centering text, drawing boxes and lines, highlighting, rearranging, and so on.

When you're happy with the layout, you pick Attributes from the main menu, then cycle through the fields assigning validation parameters. Figure 1, below, shows the extent of the options available: picture, data type, decimal precision, mandatory response, and so on. A particularly intriguing option is aux edit, which ties a user-supplied routine to a field so that you can provide validation above and beyond the capabilities offered by PC/Forms.

The test selection from the utilities menu simulates a data entry session. You can step through the form, making sure the validation criteria work and that the order of fields is right. If not, you can jump back to the editing tools and fix it without leaving FORMS. This is a particularly handy feature.

When all is well, you generate two files. One is a .FRM file, a descriptor file for the form, which we'll discuss later. The other is an application shell in generic C. It's by no means a complete application, but it contains a data structure for the fields and all the code to load and execute the form (which is only about six lines). The shell is suitable for editing and insertion into a program as a function.

The other major component of the software is a header file and a runtime library. PCFORMS.H defines the data structures, function prototypes, and what not used by PC/Forms. You include it in your source program and link the object code with the runtime library.

There are actually three runtime libraries, one for each supported C compiler (Borland, Lattice, and Microsoft). You copy the one you need from the delivery diskette. They're

```
PC/Forms Version 1,2                    ©Copyright1987 Golden Software

Single All
Assign attributres to all fields

Name: ADDRESS     Order;  4 Offset;     39 Rwo;  9 Col; 12 Len;      27

Edit Mask;
X
Default Value;
Auto Tab;           y    Right Justify;    N   Display Only;      N
Must Respond;       N    Echo Data;        Y   Upper Case;        N
Aux Edit Code;      0    Test Range;       N   Warning Only;      N
Numeric Test;       N    Data Type;        S   Pecision;          0

          Lower Bound;        0,00    Upper Bound;    0,00

[F1] Help   [F2] Display Field   [F3] Next Field  [F10] Return to Menu
```

Figure 1

Ron Copeland, associate editor for DDJ, *is the coordinator for this review section. He welcomes your feedback on products worth reviewing.*

all small models. Golden Software includes source code for the runtime system, so if you need a different model, you can recompile appropriately.

Everything your application needs in order to use PC/Forms is linked into the .EXE file. There's no separate runtime support package, TSR, interrupt-diddling, or other nuisances to clutter the environment. Only the runtime routines actually used are linked, of course, and the manual contains a table showing the code and data size for each routine. The average size is about 1K in a range of 14 to 3,974 bytes.

All identifiers have the form pcf_fname, where fname is something like "display_form" or "error." The pcf prefix makes them distinctive. It takes about half a dozen functions to load and display a form, initialize it, and get the validated input. Other functions among the 17 available do things such as altering validation attributes on the fly and releasing a form no longer needed. There are an additional 23 functions for such things as video and string management.

The .FRM descriptor produced by the FORMS editor is an ASCII file containing information specific to a given form. The runtime system needs it to implement the form and perform validation. Opening a form is a matter of loading this file. You can have several forms open at one time, and any given form can span up to ten pages (display panels). But watch out: the forms go on the stack, and you'll need a mighty big stack if you have several open at once. The FORMS editor has a utility that sizes a form and tells you how much stack space it will require.

The crucial runtime function is pcf_get_form(). When called with a form displayed, it manages user input and validation. The results are placed in a data structure corresponding to the form, whose fields can have user-assigned names. Your program then fetches data from the structure and does its thing with it. This makes the data entry portion of a loop almost ridiculously simple:

```
do {
   pcf_display_form (name, page);
   pcf_clear_form_buffer(&buf, defaults);
   pcf_put_form (&buffer);
   pcf_get_form (&buffer, &Term);
   /* then do data processing */
   while (some_condition);
```

Complaint Department

The vendor ought to include a function key template. Each function key has a purpose and some have an Alt command as well. I finally printed out the function key layout screen from the help system (which is very good, by the way).

The manual needs work. With the C version, you get the Pascal manual and a C addendum. The addendum is printed on yellow paper and you need a magnifying glass to read it because the print is so small. And there's no index, an omission that's hard to forgive even though the overall quality of the documentation is good.

These quibbles notwithstanding, PC/Forms is a real gem. It can truly save countless hours of programming, which makes it a contribution to productivity that will pay for itself many times over.

by Kent Porter

DE

Product:
DE, Version 1.2
Target:
IBM PC, IBM AT, IBM PS/2, and compatibles
Requires:
One floppy; 256K
Pricing:
$75
Vendor:
David Livshin
26 Niles Rd.
Randolph, MA 02368
617-986-7491

I don't know what DE stands for, the manual doesn't say. Were I to hazard a guess, however, I'd say it means "deluxe EMACS."

DE is a stretched version of the

(continued from page 124)

standard EMACS editor. It delivers a host of impressive features that make it a macro-programmable, customizable editor with an unlimited number of overlapping and/or tiled windows.

Unlike competing editors such as Brief, DE doesn't require a bunch of support files. It comes as a single 69K .EXE file on the delivery diskette, and installation is as simple as copying that file to your hard disk or work floppy. To invoke the editor, just type DE. Up to two command-line arguments are allowed: a -NOBAK switch to tell the program not to make a backup copy of the edited file(s), and the name of a file to be edited.

If you want to pull several files into different windows, you can fetch them after DE is up and running. The command ^X^V causes the program to prompt for a filename, then creates a new window and loads the file into it. You keep doing this until all the files you want are loaded.

Initially the windows are tiled. Each one has an information line at the bottom showing the associated filename, cursor position, number of lines, and so on, and most significantly, the window number. Various keystroke combinations let you move sequentially forward and backward among windows or jump directly from one to another. Other commands resize and drag windows so that they overlap like pieces of paper: the now familiar desktop metaphor. In overlapping mode, the current window is always on top.

If you don't like the hierarchy of windows, you can change the sequence numbering. This is a nice touch. You can make the modules you're working on neighbors in the hierarchy; it takes fewer keystrokes to move among adjacent windows than to make jumps.

The command ^X 1 does a thing called zoom and rise to the currently selected window. This changes the operation of DE by expanding each window to full-screen size and placing the current window on top. Oddly, the jump-to-next and jump-to-previous commands (^X n and ^X p) don't work any more in this mode; you have to jump to a specific window. And there's no way that I found to undo the zoom-and-rise mode. Once you're in it, you're there to stay.

EMACS commands in general are less than intuitive, and DE continues the tradition by adding still more to the repertoire. All DE commands except those dealing with editing and cursor movement begin with either ESC or ^X, followed by a keystroke denoting the command. Some make sense (^X^I to insert a file, ^X^W to write to a file, and so on), but most have no discernible connection with anything. Examples are ^_ to invoke the DOS shell, and ^X z and ^X^Z to enlarge and shrink a window, respectively.

Consequently, the vendor includes a cheat sheet showing all the keystroke commands. There's also a limited help function: type ESC ^A, then a keystroke combination, and DE tells you what function the combination performs.

Each keyboard command is mapped to a DE macro through what the vendor calls "default binding." For example, ^X^S is bound to the macro w_cfil, which writes to the current file. This association of keystrokes to macros opens the way to two features of DE: customization of the keyboard and programmability.

DE comes with 84 different macros, of which 73 are bound to default keystrokes and the other 11 (all of them related to window management) are unassigned. If you don't like the default bindings, or you want to add some bindings of your own, the command ESC ^@ runs an embedded utility that maps keystrokes to macros.

You can build your own more complex macros by combining those built into DE, thus creating editor programs invoked by a keystroke. Seldom needed programs can be stored in separate ASCII files and run with the ESC e command, which asks for the filename, loads it, and treats the contents as commands. Unfortunately, the manual barely glances at this useful feature.

Also alluded to but never explained in the manual is something called the DE.INI file. It presumably contains initialization commands that permanently map macros to keystrokes and perform other fixed set-up tasks.

It's a pity that the DE manual is not up to the quality of the software it purports to describe. A slim 26 pages, it contains terse descriptions of the macros, a little about windows and commands, and not much of anything else. The author assumes that you already know EMACS, and so leaves it to your imagination how to use the editor and its features. There isn't even a hint of a tutorial. The best part of the documentation is the cheat sheet, which puts most of the manual's contents on a card providing at a glance reference.

Overall, DE is a good editor with a lot of capability per buck.

by Kent Porter

Soft-ICE

Product:
Soft-ICE, Version 1.01
Target:
80386-based MS-DOS computers
Requires:
DOS 2.0 or later; AT BIOS
Pricing:
$386
Vendor:
Nu-Mega Technologies
P.O. Box 7607
Nashua, NH 03060-7607
603-888-2386

Soft-ICE is a product any MS-DOS developer serious enough to own a 386 machine should have. As the name implies, it provides the capabilities of an in-circuit emulator via software. For those of you not familiar with in-circuit emulators, a brief description is in order.

An in-circuit emulator (ICE) is a tool that replaces the CPU in a microprocessor-based product with a "pod" that plugs into the CPU's socket. This pod is normally connected to a box containing a control computer and some special hardware. The special hardware is used to detect user-specified conditions and to stop the processor when they occur. Another feature commonly found in ICEs is trace memory, so that when the processor stops, you can see where it has been recently. ICEs are normally expensive, and often designed more for debugging hardware rather than debugging software.

Soft-ICE gives 386 owners all of this capability, except trace memory, when debugging MS-DOS programs. It does this by using special features of the 386 normally used in writing operating systems (see February, 1988 *DDJ* for more details). Note that you can't use Soft-ICE to debug protected mode programs.

Soft-ICE can be used either stand alone or in conjunction with your favorite debugger. As a stand-alone debugger, it includes all of the necessary commands to disassemble, dump, and edit memory; to display and change registers; to peek and poke at I/O ports; and to manage breakpoints. A very useful help facility is also included, as well as a command to display the DOS system memory map. As you type commands, Soft-ICE displays a list of options.

Getting all the Breaks

Perhaps more than any other debugger, Soft-ICE lets you control breakpoints. You can set breakpoints to occur when any byte, word, or double word is read, written, read or written, or executed. For read/write breakpoints, you can include a qualifying value that must be matched or not matched. Breakpoints may be configured against a larger address range such that a breakpoint occurs on any read, write, or read or write in the range. I/O port accesses can cause breakpoints, qualified by values if desired. Execution of either a hardware or software interrupt (qualified by a value in AL, AH, or AX if desired) can cause a breakpoint. Of course, each of these breakpoints may be qualified with a count so that you can ignore the first 100 times you do any of them. You can even combine breakpoints so that a breakpoint only occurs after all of the selected breakpoints have individually occurred. Finally, breakpoints may be qualified with the location of the breaking instruction, to guarantee that the instruction is either inside or outside a range. All in all, a very comprehensive set of breakpoint capabilities, all of which can be used either stand alone or with another debugger.

If all of this isn't quite enough, you can configure Soft-ICE to generate a software interrupt when it detects a breakpoint. This user-provided interrupt gets all of the registers as they were when Soft-ICE got control, allowing the interrupt handler to do anything it likes.

When used with another debugger, Soft-ICE can be configured to trigger the other debugger when a Soft-ICE detected breakpoint occurs. It can cause an interrupt 1 or 3, or an NMI. Soft-ICE normally passes interrupt 3 onto whatever awaits it, but it can also be configured so that an INT3 returns control to Soft-ICE.

Soft-ICE is very flexible. It may be installed in normal memory, in extended memory, or in COMPAQ extended memory. If Soft-ICE finds extended memory it will automatically load there requiring none of the lower 640K. In a machine with only 640K, the program demands between 56 to 60K, rendering this segment 60K invisible to DOS.

Soft-ICE can share extended memory with other drivers, such as VDISK or RAMDRIVE. It emulates the necessary parts of the LOADALL instruction for RAMDRIVE. You can change the keystrokes used to invoke Soft-ICE. It can even boot up stand-alone code and debug it because Soft-ICE doesn't require any DOS services. Soft-ICE can also be used to debug MS-DOS device drivers. And, joy of joys, you can even debug interrupt handlers (including the keyboard interrupt) while Soft-ICE is using it.

The documentation is solid and tells you everything you need to know in about 100 pages. A tutorial chapter takes you through debugging, a simple program and is highly recommended for both beginners and those already experienced in debugging. [It does however require an IBM or equivalent for the BIOS routines. —ED]

All in all, Soft-ICE is an excellent tool for debugging 8086 programs. Compared to a true in-line emulator, (even if you have to buy a 386 machine to run it on) it's cheaper and provides superior breakpoint facilities; the only thing missing is a trace memory.

by Richard Relph

DDJ

(continued from page 14)

Compatibility Standards

Dear *DDJ*,

I've been thinking lately about how unfortunate it is that there is so little standardization in the computer industry, making it impossible to run programs written for one machine on another. I have an idea that could alleviate some of the compatibility problems, and I'd be interested in readers' responses to it.

The idea is this: Design a standard intermediate language, similar to a language that would be generated by the syntax-analysis pass of a compiler (for input to the code generator). Then, use this new language as the form in which software is distributed for use on computers, instead of as native code that is specific to a particular system. The program loader on each computer would recognize the special intermediate-language file and invoke a fast code generator to translate it to native code prior to beginning execution.

There are problems with this—one being that different systems have different capabilities and different hardware. Most systems, however, have a display device that can display ASCII characters and a printer. For many programs this is all that is needed. The intermediate language could include instructions to determine the capabilities of the system (for example, display size), allowing the program to adapt to different machines.

Another problem is that the program distribution medium is different for different systems. Even computers that use floppy disks typically each have their own disk format. This is indeed unfortunate, and manufacturers should be severely taken to task for not standardizing disk format. Perhaps it's still not too late to do this, though.

Ted Toal
Nevada City, Calif.

LAN Security

Dear *DDJ*,

It was with some dismay that I read Allen Holub's C Chest column in the February issue. Saving configuration information in the .EXE file is the easiest way I know of to discourage the use of a program on a local-area network. If users must be able to write to the .EXE file to use the program, then the program becomes a hole in system security, an opening for Trojan horse programs. Second, if users need to write to the .EXE file, multiple instances of the program cannot be run at the same time.

Often a program that has not been written with the LAN environment in mind may be used on a LAN with no modification, provided that the program may be configured dynamically. If the program may be run when its file attributes are shareable and read only, it may be possible that the program can be run on several nodes concurrently.

My preferred method is to use the traditional configuration file, but if the file is not found in the current directory, the environment should be searched for a variable named *CONFIG* that contains the path to the default configuration file. Using this method the disk does not have to contain redundant copies of the configuration file, but it can be customized for each user on the LAN through appropriate batch files.

Paul B. Hill
Norwood, Mass.

Dear *DDJ*,

Allen Holub's method of hiding configuration information in an application's .EXE file (C Chest, February 1988) is elegant and quite instructive, but I would like to raise a point or two in favor of keeping configuration information in separate files.

Most shared applications on a local-area network are located in shared public directories that are accessed via the DOS path or an equivalent network function. Users of the shared applications are usually granted read-only access to these public directories. Most users cannot be allowed to modify files in the public areas as this would be an invitation to disaster. Application programs that write into their own .EXE file do not work well under these circumstances and can cause LAN managers severe headaches. The re-

sult is that you end up with several complete copies of the application program scattered about.

It is possible for a LAN manager to arrange for the proper storage of an application's configuration file if the program is designed with some thought to the problem. I believe the best way to handle the location of the configuration file is for the application to accept a command-line parameter that points out the complete path where the configuration file is located.

The LAN manager usually has methods of setting environment variables when users log on to a LAN that can specify user names of physical workstation numbers. These environment variables can then be used with shared public batch files to execute the application and specify the appropriate configuration file for a particular user or workstation. It is useful to be able to specify a configuration file in association with a physical workstation because most networks have a variety of workstation hardware.

The programs that are most convenient to install and use on local-area networks are those that keep things simple. The best applications are often those that consist of a single .EXE file with no other external files. These are also the programs that LAN managers will purchase in quantity rather than those applications that cause difficulties in configuration, installation, and maintenance. Most of the software currently used on LANs are not the large expensive packages that do strange tricks with configuration and security information but the same simple single-user products that work so well on Personal Computers.

Phillip M. Nickell
Longmont, Colo.

Was He Misguided?

Dear *DDJ*,

I was happy to see the review of The Norton Guides (Examining Room, February 1988); however, I was reminded of these words by Lewis Mumford in the *Pentagon of Power*: "Unfortunately, 'information retrieving,' however swift, is no substitute for discovering by direct personal inspection knowledge whose very existence one had possibly never been aware of, and following it at one's own pace through the further ramification of relevant literature. But even if books are not abandoned, but continue their present rate of production, the multiplication of microfilms actually magnifies the central problem—that of coping with quantity—and postpones the real solution, which must be conceived on quite other than purely mechanical lines: namely, by a reassertion of human selectivity and moral self-discipline, leading to continent productivity. Without such self-imposed restraints the over-production of books will bring about a state of intellectual enervation and depletion hardly to be distinguished from massive ignorance."

As a reasonably happy owner of The Guides, I eagerly dug into the review, which upon reading, I'm afraid to say seemed like a cursory tidbit that I had not expected to find in *DDJ*. I feel there are three major problems with it:

1. I originally thought that the 37K memory residency TSR requirement was a gnomish munchie, but after looking at the 38K file size on my floppy, I thought there was definitely a mistake here. I have run this program on both an IBM PC and an IBM AT compatible with PC-DOS/MS-DOS, and it takes up 71,920 bytes installed.
2. One of the original problems with the advertising, and especially on the box my program came in, was the reference to being able to run it on a floppy-disk-based system. This is possible if you have drives that hold more than the 600K needed for either the MASM or C database. A hard disk is probably required rather than recommended, unless you wish to write your own or just use a BASIC database. I was not pleased to find that the assembly-language database would not run on my IBM PC floppy system. The review made no mention of this possible predicament.
3. One of the most useful aspects of the program is the capability of The Guides (given the right active menu) when initially activated to do an automatic lookup of the entry for the word by the cursor. It's a handy feature to have enabled but nary a mention of it in the review.

I would think that if you're going to do these reviews, you need to have people to do them who have more than a passing interest in the material. As the ultimate end-user of some of this software, I'd like to get information other than what I can get from ads and the users' booklet. I'd also like to see how the product compares with others that purport to do the same thing but cost less.

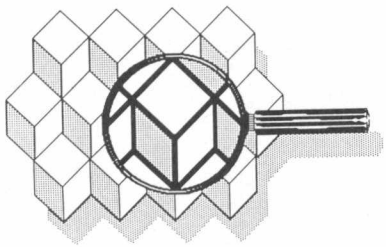
Richard L. Henley
via CompuServe

Kent Porter responds:
Let's go by the numbers:

1. We're both wrong according to CHKDSK; it takes 72,032 bytes on my AT. The NG.EXE file is 37K, not RAM resident.
2. Good point. I haven't run it on a floppy-based system, but Richard's point makes sense because the databases are large.
3. Auto-lookup is indeed a dandy feature. If it got short shrift, it's because space in these reviews is limited.

DDJ

OF INTEREST



Products for Developers

Dan Bricklin's Demo II Program is now available from **Software Garden**. This new version comes with a 220 page manual, keyboard templates, an on-line tutorial, and sample files. New features include the ability to capture bitmapped graphics images from other programs, string and numeric variables, and a run facility with over 100 new actions to execute while running. Demo II also comes with a license to make an unlimited number of copies of the runtime. The product runs on 512K IBM PC, IBM PC AT, IBM PS/2 or compatibles using DOS 2.0 or later. A monochrome display adapter, CGA, EGA, VGA, Hercules Graphics Card or the equivalent is also required. Demo II sells for $195. Reader Service No. 20.
Software Garden Inc.
P.O. Box 373
Newton Highlands, MA 02161
617-332-2240

National Design Inc. (NDI) has released Genesis 1024 and Genesis 1280 intelligent PC color-graphics controllers bundled with an implementation of the Computer Graphics Interface (CGI) developed by Nova Graphics International and Metagraphics' MetaWINDOW.

The NOVA*CGI provides software developers a CGI interface that is now resident on the NDI controllers. By executing the CGI on the NDI board, the developer can perform graphics routines 20 to 40 times faster than on a PC using an EGA card.

The Genesis 1024 (a 640 × 480 up to 1,024 × 768, 16 color card), and the Genesis 1280 (a 640 × 480 up to 1,280 × 1,024, 256 color card) use Texas Instruments' 34010 graphics system processor operating at 40 or 50 MHz. The Genesis products also provide expandable memory up to 32 Mbyte.

The Genesis 1024 costs $1,700 and the price of the Genesis 1280 ranges from $2,995 to $3,995, depending on configuration. Reader Service No. 21.
National Design Inc.
9171 Capital of Texas Hwy. N.
Houston Bldg., Ste. 230
Austin, TX 78759
512-343-5055

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie has been published by **Prentice Hall**. This new edition is based on the draft-proposed ANSI C Standard. The book makes precise the features that were not spelled out in the original definition of C, and states explicitly which aspects of the language remain machine dependent. New features from the ANSI standard, such as function prototypes and the standard library, are also explained. Additional changes in the new edition include a C reference manual, an appendix describing the standard library, and an appendix summarizing changes between the first edition and the draft-proposed ANSI standard. The price of the book is $40 for cloth and $28 for paper. Prentice Hall will continue to publish the first edition. Reader Service No. 22.
Prentice Hall
Prentice Hall Bldg.
Englewood Cliffs, NJ 07632
201-592-2000

The Software Link (TSL) has announced its newly formed Developer Relations program. The program is based around the company's newly released PC-MOS/386 Technical Reference Manual. Participants subscribe for an annual fee of $500 to development support service which includes: PC-MOS/386 Technical Reference Manual and updates as they become available, access to TSL's support line, upgrades of PC-MOS/386, participation in TSL's product certification program and inclusion in TSL's product reference guide, and purchase of one PC-MOS package for development use at a reduced rate. Reader Service No. 23.
The Software Link
3577 Parkway Ln.
Norcross, GA 30092
404-448-5465

World Wide Data has released Charm, a C source application generator. Charm is an integrated application generator for Unix and VMS environments that automatically creates fully documented C source code. Charm's 4GL, dali (data access language interface) is a natural extension of the interactive screen and program generator. All the standard field default and verification options in Charm are dali programs. Reader Service No. 24.
World Wide Data
17 Battery Pl.
New York, NY 10004
718-438-2807

The Renaissance Graphics Device Interface (RGDI) Developer's Kit is a toolkit for software developers that enables them to develop graphics applications that take advantage of the speed of a special graphics processing chip. **Renaissance GRX's** new product includes: Rendition I Advanced Graphics Controller incorporating RGDI, Rendition I user's guide, RGDI programmer's technical reference manual, TI 34010 user's guide, and development software.

The RGDI is a graphics controller interface that allows a software program to send messages to the Texas Instruments TMS34010 Graphics System Processor, a 32-bit, high-speed integrated circuit that is optimized for graphics performance.

For developers who wish to write in assembly language, Renaissance offers an optional accompanying advanced toolkit which includes: TI 34010 debugger and user's guide; TI 34010 assembler package, including an assembler, linker, and simulator; and development utilities.

For developers wishing EGA compatibility, an optional Rendition EGA

(REGA) plug-in model is available.

The RGDI Developer's Kit is available for $695. The optional Advanced RGDI Developer's Kit add-on is priced at $495. The REGA option costs $169. Reader Service No. 25.
Renaissance GRX
Cedar Park
2265 116th Ave. NE
Bellevue, WA 98004
206-454-8086

SoftScience Corp. has released its Convenience Plus DOS Front End which is designed for use with IBM's new 3363 Optical Disk. The program is intended for the novice and advanced PC user as a front end to MS-DOS, PC-DOS, and OS/2 and is designed for supporting file management and recovery on the IBM 3363 Optical Disk.

The program features the ability to perform DOS commands and additional commands not available through DOS; the ability to organize and understand the arrangement graphically of the computer; the ability to use the computer without memorizing or typing complicated syntax and language at a faster pace; unusual file recovery and management commands for the IBM Optical Disk; and DOS commands in five foreign languages. Reader Service No. 26.
SoftScience Corp.
Box 42905
Tucson, AZ 85733-2905
602-326-4679

Sourcer, available from **V Communications**, allows programmers to create commented source code and listings directly from RAM, ROM, .COM files and .EXE files. Sourcer creates detailed commented listings and source code directly suitable for assembly. Built-in data analyzer and code simulator resolves data items across multiple data segments, provides detailed comments on BIOS and MS-DOS interrupt calls and subfunctions, and I/O ports. Sourcer also determines proper assembler directives for multi-segment programs. Built in processor filter optimizes code based on instruction set selected, 80286, 80186/88, 8088/86 and V20/V30.

The Sourcer is also available with the BIOS Pre-Processor, which provides the first means to obtain accurate legal source listings for any BIOS. It identifies entry points with detailed in-line comments explaining functions and subfunctions, registers, and other key information.

The Sourcer costs $99.95, the Sourcer with BIOS Pre-Processor costs $139.95. Reader Service No. 27.
V Communications
3031 Tisch Wy., Ste. 200
San Jose, CA 95128
408-296-4224

Blaise Computing has announced ASYNCH PLUS/4.0, a comprehensive set of routines designed for Turbo Pascal 4.0 to give programmers the power to create interrupt driven communications software. The program has a layered design of separately compiled units, with the higher levels building on the lower levels. These routines drive virtually any asynchronous device via the RS-232 ports. ASYNCH PLUS includes low level control and queue maintenance functions written in assembly language and high-level routines written in Turbo Pascal to help programmers develop communication software. Fully documented source code is included as well as a comprehensive indexed manual which gives a general overview for every function category and descriptions of each function. Examples in the manual and full programs on the distribution diskettes serve as illustrations. ASYNCH PLUS is priced at $129. Reader Service No. 28.
Blaise Computing
2560 Ninth St., Ste. 316
Berkeley, CA 94710
415-540-5441

Programs in Motion offers a shortcut to large-system expert system developers by parlaying its facile, direct handling of decision trees with its new ability to generate production rules in a variety of programming languages. This expert system development software can work as a scratch pad for quick decision-tree prototyping, or even as a working breadboard. Once a decision tree functions as desired, the software

can generate corresponding production rules of program code in C or Pascal. In most cases, these modules can transfer into big expert system environments with little or no change. 1st-Class Fusion for IBM PCs, IBM XT, IBM ATs, or compatibles is priced at $1,295. Reader Service No. 29.
Programs in Motion
286 Boston Post Rd.
Wayland, MA 01778
617-358-7722

The GSS Graphics Development Toolkit for OS/2 is available from **Graphic Software Systems**. The GSS Graphics Development Toolkit for OS/2 provides a high-performance graphics development environment for OS/2-based personal computers. Its high-level functions speed development of interactive graphics applications, and a growing set of device drivers removes the burden of writing driver code for input, display, and output devices. The Graphics Development Toolkit supports advanced features of OS/2 and maintains source code compatibility with the Graphic Development Toolkit for DOS.

The GSS Graphics Development Toolkits for DOS and OS/2 are priced at $495 and $695, respectively. Reader Service No. 30.
Graphic Software Systems
9590 SW Gemini Dr.
P.O. Box 4900
Beaverton, OR 97005
503-641-2200

Tools and Utilities

Flambeaux Software has announced TECH Help! Version 3.3A. TECH Help! is an on-screen reference for system-level programmers. It includes coverage of the DOS and ROM-BIOS services, system variables, I/O ports, installable device drivers, and the layouts and structures of dozens of data tables, bit flags, and switch settings. It covers some topics which are not documented in the official reference manuals. The new version covers DOS 3.3 and the latest versions of the ROM-BIOS.

The display driver has been upgraded to include user-configurable color selection, a simplified way to access multiple Help! manuals, and increased display speed for EGA and VGA monitors.

TECH Help! runs on computers that are compatible with the IBM PC, IBM XT, IBM AT, and IBM PS/2 computers. The program is priced at $89.95. Reader Service No. 31.
Flambeaux Software
1147 E. Broadway, Ste. 56
Glendale, CA 91205
818-500-0044

California 10 PAK, by **California Software Products**, has been upgraded to be used with OS/2 on the IBM PS/2 and compatible machines. California 10 PAK contains 16 programs for browsing, comparing, and sorting the contents of files and memory. System configuration and a map of all installed memory may be displayed. A disassembler produces ready-to-edit-and-assemble source files from .COM files, .EXE files, or any area of main memory. An operating system shell allows users to define the operation of function keys and to create color menus and help screens. California 10 PAK runs under any version of DOS and under OS/2 in protected or unprotected modes. The price for the product is $79. Reader Service No. 32.
California Software Products
525 N. Cabrillo Park Dr.
Santa Ana, CA 92701-5017
714-973-0440

DDJ

SWAINE'S FLAMES

Everything is deeply intertwingled.
—Ted Nelson

I keep reading things that remind me of Ted Nelson's *ComputerLib/Dream Machines* (either the original landmark manifesto à la collage of 1974 or the radically revised and recently rereleased Microsoft Press edition). There are several possible explanations.

First, Ted talks about a lot of things in *CL/DM*; his is an eclectic light.

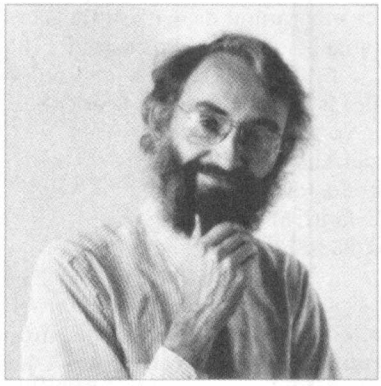
Second, he mostly writes about things I mostly read about anyway: the potential of computers, the art of writing, the liberation of the mind.

Third, we remember unfinished business better than finished, and *CL/DM* is a business of unfinishedness; of loose ends; boxes that, once opened, can't be closed again; conceptual gambits; twitching, severed nerves; choose your metaphor. *CL/DM* pitches more itches than it hawks ointments for.

Fourth, everything is deeply intertwingled. That's probably it.

One thing that I've been reading that implicitly invokes *CL/DM* is *The Society of Mind* by Marvin Minsky. *The Society of Mind* presents a model of the mind as a society of communicating processes. The structure of the book reflects Minsky's model and reminds me of *CL/DM*. If you haven't read it, I recommend it.

I've also been reading about highly functionally distributed systems (HFDS): shades of Ted Nelson's *System Xanadu*. An HFDS is a heterogeneous loosely coupled worldwide network of computers and other "intelligent" objects, providing more sophisticated services than any of its components can, says University of Tokyo professor Ken Sakamura, who named HFDS. The multinational, multicompany project Sakamura spawned to implement it has a better name: TRON.



The TRON project envisions a global network and several types of networked devices, including intelligent objects and communication machines. The TRON team is designing a system for the day when a typical room contains a hundred computers, a building thousands, a city millions. The problems that would arise include questions like just how much should my neighbor's air conditioner know about my new lamp?

In an HFDS, neither centralized control nor anarchy would work. The cooperation of groups of components in an HFDS Nakamura intertwingledly calls the "society of computers."

The elements of the TRON project include ITRON, a spec for a real-time operating system for the control of intelligent objects; BTRON, an operating system spec for the human interface components of an HFDS; CTRON, a portable operating system spec for servers that will link BTRON and ITRON elements with gateways and large databases; and MTRON, which somehow ties it all together. The idea is that ITRON talks to machines, BTRON talks to people, CTRON talks to ITRON and BTRON machines, MTRON talks to Matsushita, and Matsushita talks only to God. I think.

Last November, TRON developers from more than 100 firms, including Matsushita, Fujitsu, AT&T, and IBM, met to report progress. Intended to go on-line in the 1990s, TRON is ahead of schedule.

I've been reading a lot about Apple's HyperCard lately, too, since I'm writing a book on it. The "hyper" is homage to Ted's hypertext, although he has some reservations re HyperCard as an implementation of same.

Others have also raised doubts about the market for HyperCard stackware. A panel at January's MacWorld show, discussing stackware prospects, was not encouraging. "Not until there are 100,000 CD-ROM units installed will there be a decent stackware market," one panelist concluded. Stewart Alsop thinks that unless developers, distributors, and Apple all treat stackware like full-fledged serious software, "The stackware business will disappoint just as many people as the 1-2-3 and dBASE templates business did." And Dvorak declares, "I wish those budding stackware developers a ton of luck. They'll need it."

I suspect that the world will little note nor long remember such sage skepticism, even if it turns out to be true. Since Apple brought forth on this continent the computer for the rest of us, Macintosh software development has been beyond the poor powers of unprofessional performers, but now any Sunday afternoon matinee walk-on bit player can prototype a product in a day or two, down to doing a cover letter in card form and printing the disk label.

Heaps of hyperstuff will be hacked together. Publishers will be pummeled in submissions if not into submission, and rejection will be no deterrent to the hyped-up hordes of hyperdom.

Reminds me of what Ted Nelson said about all that white-out on the screen. . . .

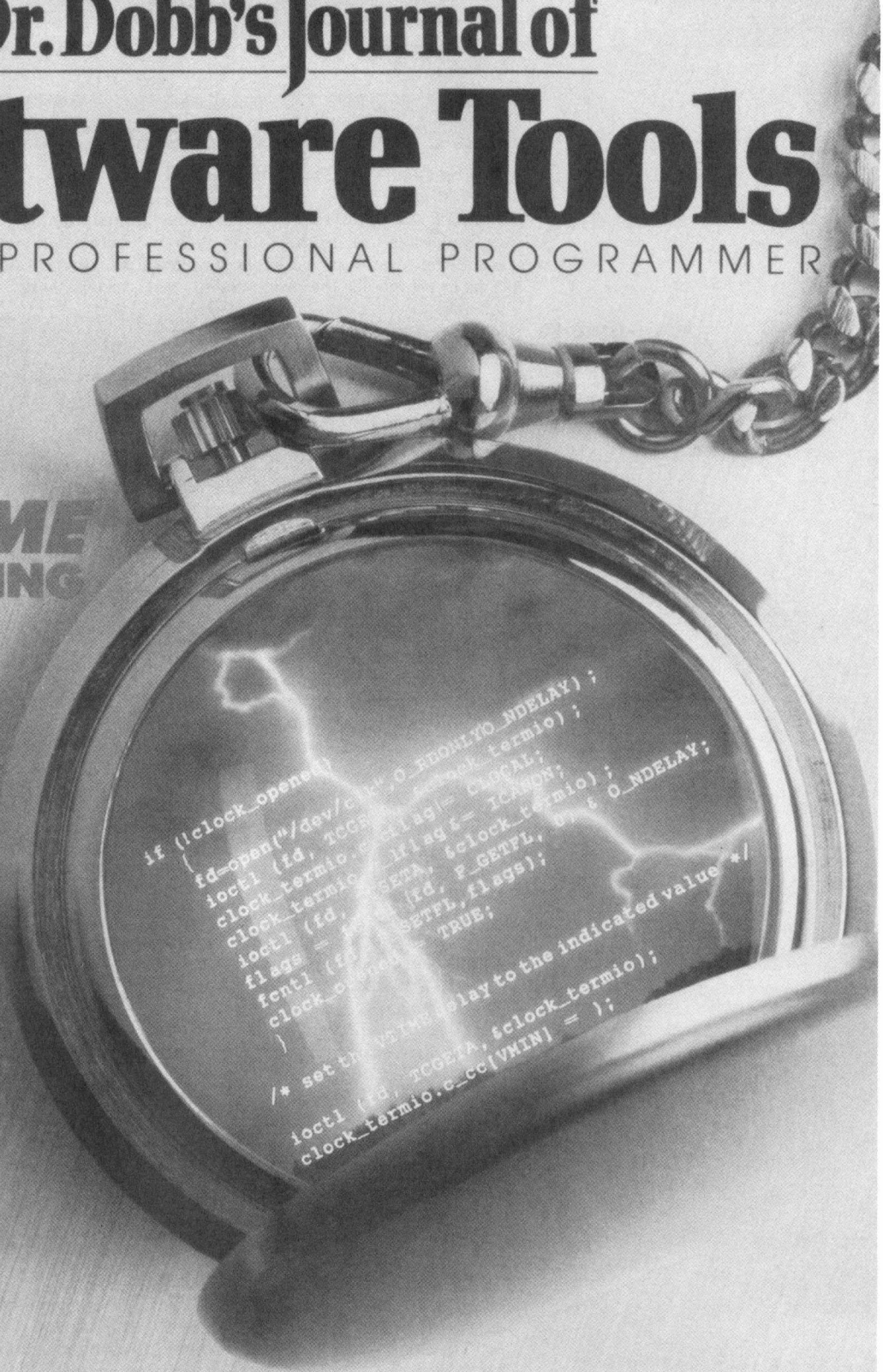
Michael Swaine

#140 JUNE

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER



REAL-TIME PROGRAMMING

RECREATING SOURCE CODE

TRACKING LOST POINTERS

PRODUCTS EXAMINED

- C-Index+
- MS Pascal 4.0
- Peabody for Turbo C

LANGUAGES: C, Pascal, Forth

0 38351 16562 8 06

ARTICLES

Real-time ▶ **Writing Real-Time Programs Under Unix** **18**
by Bill Cramer
Porting a real-time process to a Unix environment isn't always an impossible task. In this article, Bill offers some guidelines and C routines to let you either simulate a real-time OS with Unix, or reorganize the program with a message-passing scheme.

Real-time ▶ **Message-Passing Operating Systems** **34**
by Dan Hildebrand
An overview of operating system design using intertask messaging, with real-world examples from QNX.

A Simple Decompiler **50**
by William May
Bill May returns to *DDJ* with a simple algorithm for recreating source code from tokens. The example source code is in C.

REVIEWS

EXAMINING ROOM **124**
coordinated by Ron Copeland
Products examined from the programmer's perspective. This month: The Peabody online programmer's reference, the C-Index+ data file management library, and T-Debug Plus 4.0.

COLUMNS

C Pointers ▶ **C CHEST** **80**
by Allen Holub
Allen tackles some of the nastiest bugs around with a set of subroutines for tracking down pointers that muck with *malloc( )* and *free( )*.

Forth ▶ **THE FORTH COLUMN** **90**
by Martin Tracy
Martin offers some thoughts on the state of real-time Forth, as well as source code for floating-point extensions.

Pascal ▶ **STRUCTURED PROGRAMMING** **98**
by Kent Porter
Kent examines the new release of Microsoft's Pascal compiler and compares it to Turbo Pascal 4.0; he also takes a short, painful look at Intel's LIM 4.0 upgrade.

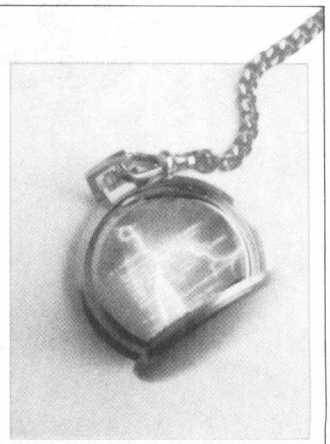
Transputers ▶ **PROGRAMMING PARADIGMS** **110**
by Michael Swaine
Notes on objectivity at Software Development '88, as well as Prolog, parallelism, and transputers.

FORUM

EDITORIAL **6**
by Jonathan Erickson
RUNNING LIGHT **8**
by Tyler Sperry
LETTERS **12**
by you
CARTOON **14**
by Joe Sikoryak
SWAINE'S FLAMES **152**
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST **140**
brief product descriptions
ADVERTISER INDEX **145**
Where to go for more information on products.
PROGRAMMER'S MARKET PLACE **148**
classified ads



About the Cover

No jokes, please, about code so ugly it could stop a clock. The pocket watch was *our second* idea: We'd originally planned to do an inner view of a task scheduler for the cover, but it didn't work because we were always getting interrupted.

Next Issue

July will see us covering the frontiers of databases: including some hard information on programming for SQL environments and other goodies.

EDITORIAL

It's becoming clear that one of the more pressing problems facing software developers will be the ability to find enough qualified programmers to write the software that needs to be written. According to many companies we've talked to, there's a shortage of good programmers right now and the indications are that the situation won't be getting any better.

A couple of recent conversations brought this to mind. In one instance, the president of a major software company mentioned to us that he is getting much of his programming done in Mexico because he can't hire enough programmers in this country. (The fact that he can hire experienced programmers in that country for one-fifth of what it costs in the U.S. is also a motivating factor, something he conceded only when we challenged him about it.) In another conversation, General Bill Thurman, commander of the Air Force's Aeronautical Systems Division, told us that recruiting, training, and keeping qualified programmers is becoming a problem of major proportions.

Compounding the problem is the simple fact that enrollments in university computer science programs continues to dip. For the first time in years, smaller colleges have openings for prospective computer science majors. Larger universities are still filling their slots, but they are seeing a dramatic decrease in interest in computer science. Last year, for instance, the University of California at Berkeley had 11 applicants for each computer science opening. Next year, one university spokesman told us a few days ago, the university expects each opening to have as few as five applicants.

Although the doctrine of supply-and-demand may provide short-term benefits for many programmers, the long-term consequences might be more detrimental. If nothing else, the pressure necessitated by competitive schedules will have an adverse impact on the sanity of many programmers. That's the bad news. The good news is that programmers will be in a better bargaining position when it comes to defining their workplace because those companies that are able to attract top-flight programmers will be the companies who get the best software products out first.

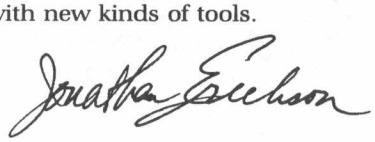
Working in real-time environments usually requires specialized development tools and sometimes specialized operating systems. The variety and complexity of real-time operating systems, however, may be foreign to those used to working with more mainstream, conventional operating systems. (The main distinction between real-time operating systems and more familiar one is in the way the real-time OS handles the task scheduling and priorities that enable predictable response time to events.)

We're of the opinion that real-time, embedded system development is an area with exciting potential. One reason for believing this is reflected in the growth in the embedded controller market itself.

By 1991, it is estimated that more than 800 million controllers will be shipped annually.

In practical terms, this has led to real technological innovation.

These emerging hardware platforms will give programmers the chance to create all kinds of new applications, but those programmers may first have to become familiar with new kinds of tools.



Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editors *Rhoda Simmons, Carol Dondrea, Kevin Shafer*
Editor-at-Large *Michael Swaine*
Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Barbara Mautz*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Asst. Circulation Manager *Andrea Weingart*
Newsstand Manager *Sarah Frisbie*
Direct Marketing *Kathleen Shay*
Fulfillment Coordinator *Francesca Martin*
Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, with two special issues per year by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA 800-321-3333; in CA 619-485-9623 or 566-6947. For book/software orders call 415-366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

I've written before in this space about the tyranny of time. Not about real-time, mind you, but about deadlines and schedules and the three month schism between magazine time and reality, and how that makes most editors a little wonky by the time they close an issue. Given this shared background, I trust you'll understand when I spend a little time talking about this year's West Coast Computer Faire, an event that is a week past for me, and at least two months past as you read this. As usual, the most interesting things are signposts for what's up ahead.

As either punishment for past sins or undeserved good fortune (I'm still not sure which) I was roped into being the moderator of the "Language Wars" panel. The audience looked to be your average computer faire crowd—just average folks, dressed in T-shirts, jeans, and sneakers. Make that your average *programming* folks, since over half the audience said they programmed for a living.

The panel speakers included Ray Duncan (ex-*DDJ* columnist and friend of the *DDJ* family), Jim Anderson (from Digitalk), Greg Lobdell (from Microsoft), and David Intersimone (from Borland). Given that the majority of the panel members were language pluralists, and that the audience seemed evenly divided into Basic, C, and Pascal camps, we didn't spend much time on the religious issues of picking the "One True Programming Language." Instead, we spent most of the afternoon discussing language and implementation features—what programmers want and need in compilers. Here are some of the highlights:

- Microsoft had just conducted some extensive marketing research on the QuickBasic users. They weren't quite expecting the results they got: over 30 percent of the QB programmers surveyed were using the product in the primary job function. So much for the death of Basic.
- David Intersimone stressed the importance of debugging in his talk. Although he wouldn't make any "official" announcements of products, it's a safe bet we'll see integrated debugging from Borland within a month or two of this issue.
- Jim Anderson did spend some time talking about the advanced debugging his company was putting into Smalltalk/286, the new protected mode version of Smalltalk/V that should have shipped last month as you read this. (I warned you that timing was tricky, didn't I?) As soon as the Digitalk crew polishes off the 286 version, they'll be putting the same debugging tool into a new product, Smalltalk/Mac.
- Your fearless editor cautioned the audience not to confuse the merits of a particular language with the merits (or failings) of a particular implementation. Specifically, I noted that despite C's supposed "inherent efficiency," the latest crop of Modula-2 compilers were surpassing both Borland's and Microsoft's C compilers in benchmarks. Look for more on Modula-2 in Kent Porter's column in the coming months.

As usual, you can reach me online with your comments on languages, suggestions, and article ideas in a number of ways: on BIX as "tyler," on GEnie as "T.SPERRY," and on CompuServe with the catchy number 76703,4266. Those of you enamored with the Unix system and uucp will be pleased to learn that *DDJ* is finally online, thanks to the efforts of Phil Sih down at Portal Communications in Cupertino. Unix mongers can now reach me as sun!cup.portal.com!tyler.

Tyler Sperry
editor

ARCHIVES

Ten Years ago in *DDJ*

"Many unfortunate souls have experienced the frustration which occurs when you realize that you've just accidently erased the program you were working on. Kicking the computer may provide some emotional satisfaction but it won't bring your program back."—Andy Hertzfeld, "Lazarus–*A Program to Resurrect BASIC Programs on the Apple-II Computer*," DDJ, *June 1978.*

Left to your own devices

"Forth allows the programmer unrestricted access to the underlying hardware, but offers almost no protection against misbehaving programs. Thus, the computer reset button is an indispensable tool in Forth program debugging. . . ."—N. Solntseff, "*Forth Programming Style*," DDJ, *September 1982.*

Nonpareil

"True parallel processing involves both a rethinking of system architecture and a fundamental recasting of algorithms. It exists. There are parallel-processing computers in existence today, and more are on the way. Full use of those computers—exploitation of parallel processing at the level at which we have exploited sequential processing today—represents a large step forward, but it's not here yet. It's a level 50 problem in the notation of Knuth's Fundamental Algorithms. General-purpose parallelism in machine architecture and algorithm design may well require cooperation on the level of the efforts that produced the sequential machines of the 1940s. That cooperation is not always present."—Michael Swaine, "*The Problems of Parallelism*," DDJ, *March 1986.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Pascal, C, and Row Storage of Arrays

Dear *DDJ*,

In his "Structured Programming" column in the March 1988 issue, Kent Porter points out that two-dimensional arrays can be implemented as an array of pointers to rows instead of the usual contiguous array of rows (where a row is itself an array of data elements). Besides overcoming the 8088 64K segment restriction, this programming technique is commonly used for arrays where the rows have different lengths or are of uncertain length and number.

The best known example for C programmers is the processing of the command line. The hidden startup code splits the command line into words and allocates space for and fills in an array of pointers to the words. It then passes the address of the pointer array to the function *main( )*. Other examples are given in Kernighan and Ritchie's *The C Programming Language*.

Porter's implementation of this technique in Pascal, while a service for Pascal programmers, also illustrates some contrasts between Pascal and C. In Pascal, array element references change their syntax, after auxiliary type declarations, from a^(I,J) to the more awkward A^(I).col^(J). To convert a program to use large matrices with the alternate storage method, all array references would have to be modified. In C, only the allocation function would have to be changed. Arrays references would keep the form A[I][J] in spite of the change in storage method since the compiler handles such internal details. This is illustrated by the following short program which prints the command line as a two-dimensional array of characters.

```
main(int n, char **v) {
   register int i, j;
   for (i=0; v[i]; i++) {
   for (j=0; v[i][j]; j++)
   putchar(v[i][j]);
   putchar('\n');
   }
   }
```

Admittedly, however, if the C programmer had used explicit pointer references for the sake of 'efficiency,' then the conversion effort might be even greater than with Pascal.

Terry J. Reedy
Los Angeles, CA

Fortran Implementation of Hugemats

Dear *DDJ*,

I realize that Fortran is no longer fashionable as a computer language, but—without casting aspersions on Borland's dialect of Pascal—I would like to submit the example below as a substitute for Kent Porter's Turbo Pascal program hugemats (*DDJ*, March 1988). Apart from the compiler directives, it is a portable Fortran 77 program and, although without comments, its intent is more immediately obvious.

```
c Fortran implementation of ''hugemats''

c

      program hugemat
      implicit integer*2 (a - z)

      parameter (mrow=250, mcol=300)

      dimension a(mrow,mcol), b(mrow,mcol),
                              c(mrow,mcol)

      call acquir (a, mrow, mcol)
      call acquir (b, mrow, mcol)

      do 20 i = 1, mrow
         do 10 j = 1, mcol
            c(i,j) = a*i,j) + b(i,j)
 10      continue
 20   continue
      print 30, 'Proof:'
      print 40, 'l[1,1]', a(1,1)
      print 40, 'b[1,1]', b(1,1)
      print 40, 'c[1,1]', c(1,1)
      print 30, ' '
      print 40, 'a[mrow,mcol]', a(mrow,mcol)
      print 40, 'b[mrow,mcol]', b(mrow,mcol)
      print 40, 'c[mrow,mcol]', c(mrow,mcol)
 30   format (' ',a)
 40   format (' ',a,' = ',i5)
      end

      subroutine acquir (array, n1, n2)
      implicit integer*2 (a - z)
      dimension array(n1,n2)

      do 20 8 = 1, n1
         do 10 j = 1, n2
            array(i,j) = i * 10 + j
 10      continue
 20   continue
      return
      end
```

Example 1: *A Fortran Implementation of "hugemats"*

C.B. Chapman
London, Ontario

And Debate Goes On

Dear *DDJ*,

I have followed the "Turbo C vs. Quick C" debate with some interest. Availability? Quick C has been on sale at my local branch of B. Dalton for weeks. Turbo C was rushed to market earlier (but still late) and full of bugs.

Few people have focused upon the single most important difference between the two compilers—debugging. In my many years as a professional programmer I have learned that finding runtime bugs is by far the most difficult and time consuming part of programming. Finding syntax errors at compile time is normal, but not significant. Why has Borland never produced a source-level debugger like CodeView? There are plenty out there that Borland could simply buy and customize. I think that this is the single greatest weakness of all Borland products.

Colin J. Davies
Granada Hills, CA

Dear *DDJ*,

I was reading with amusement your recent exchanges about the relative merits of the Quick C and Turbo C compilers. Since I have bought and used both, and since I returned my Quick C compiler for a refund, I have some opinions about them. My version of Quick C was 1.0, and my Turbo C is 1.5. I have a 4,800 line program that had already trashed my Let's C source-level debugger and my DOS-Debug utility, and it did the same for Quick C's debugger. The Quick C debugger developed a case of frozen screen with funny-looking symbols and music notes appearing at random on what

(continued from page 12)

was supposed to be the display of my program executing.

About the Quick C documentation: Yes, indeed, it does have excellent C documentation, although the Turbo C documentation really is not that bad compared with, say, *The C Programming Language* of K and R. But the documentation for the Quick C debugger is almost nonexistent and what is there is cryptic and unusable. Let someone cleverer than me find out how to single-step the debugger from the documentation; or, how to stop the debugger once it is started in "display steps in execution" mode. There are no answers anywhere in the manuals, nor even references in the index on the debugger.

Since the main reason that I bought Quick C was to use the debugger, I have some reasons to dislike what I got: first of all, the debugger is crippled. It only accepts programs that have just been compiled, and excludes re-loading and debugging previously compiled programs. You can only compile in medium model if you want the debugger to work, even if you have a program that only works in huge model (I hope C beginners realize that large programs have to use special "memory models" to work properly on the PC), and Quick C does not support huge model compilation. And, let me tell you, you can write a will and fill out your income tax while Quick C is compiling and then loading the debugger. With my 4,800-line program, the debugger took over a half hour to compile and load. If you have ever used the Let's C compiler and source debugger, you know that Quick C makes Let's C look good on that basis.

Do you know what the underground bargain C compiler of this year is? It's the Mix Power C compiler. For under $25 with shipping, it is one heck of a good compiler, and the manual is worth the price of admission. It's C preprocessor will substitute macro arguments into the middle of strings for you (so you can macroize your *printf* ("debug message with dubbed parameter") statements using your *h* file); and, while it has a problem with *newlines* inserted in the middle of parameters supplied to the macro-processor, it is otherwise quite stable and handles really large programs (in large model, not huge). While Turbo C is faster and can handle huge model programs, its macro preprocessor will not substitute arguments into the middle of strings for you, instead the strings resulting from the macro-expansion will contain the macro parameter name.

To be fair, the Turbo C Windows package is worth the cost of the compiler, and the Power C business math and graphics package is definitely a worthwhile acquisition. Turbo C has to be the world's fastest screen writer, both in windows mode and standard *printf* to *stdout* and *stderr* mode. Both Turbo C and Power C have optional run-time stack overflow tests automatically inserted into compiled code.

Since C programs work in mysterious ways and can and do overwrite memory and go into recursive bugs, the stack test is a minimal safety net that is really needed. I only wish that the enumeration type of C could be used to force range tests Pascal-style in compiled code. That way, while debugging I could *enum* some of my important variables and specify their lower and upper bounds in the *enum* set, knowing that the compiler would insert correctness tests wherever I did C arithmetic with these variables. As matters presently stand, the compilers refuse to let me do arithmetic with these variables, and don't seem to enforce bounds on their values by run-time checking.

Victor Schneider
Brighton, MA 02146

DDJ

Unbeknownst to his colleagues, Gil had reached the end of his interrupt chain.

WRITING REAL-TIME PROGRAMS UNDER UNIX

by Bill Cramer

For most programmers, the expression "real-time with Unix" falls into the same category as "user-friendly yet powerful" and "self-documenting code." But just as there actually exist a few powerful, user-friendly products (which were no doubt written with self-documenting code), Unix is a realistic and practical operating system choice for many real-time or near-real-time applications.

Most applications fall into one of three broad classes—interactive (such as a word processor), batch mode (such as a payroll program), and real time (such as the module in a car that tells you the door is open). These applications perform best when running under an appropriate operating system and perform worst when running under an inappropriate OS—for instance, a payroll program that requires extensive user interaction for each of 19,000 employees would drive an accounting staff over the edge. Likewise, using a batch-mode operating system to read the temperature module on your car's engine might not tell you that the car has overheated until a few hours after the fact.

A few applications, however, need a subtle mix of different operating system characteristics. In a computer-automated assembly line, for example, each assembly workstation may have its own computer or embedded processor running under a real-time operating system. These real-time computers keep busy reading gauges, moving robot arms, turning switches on and off, and so on. A central computer that doesn't need to precisely control valves opening or the amount of torque a robot arm applies to the handle may preside over the individual workstations. That central computer has more global concerns—for instance, it may have to shut down the assembly line if an individual station doesn't have the parts required to build the product.

The central computer may intermittently receive input (how many parts it has processed and so on) from individual workstations. It may then send commands to other workstations to speed up or slow down the assembly line or to reroute a certain part to a different station. It may also sound an audible alarm to alert the foreman that a particular machine needs servicing.

The real-time OS in the workstation controller generally performs the work it was designed to

Bill Cramer is a software engineer with Teknekron Infoswitch, 1784 Firman Dr., Richardson, TX 75081, which builds Unix-based adjunct processors for PBX and central-office telephone switches. Most recently, he has been working in the area of software productivity.

(continued from page 18)

do very well; however, it was never really intended to write reports, maintain a database, or display the status of all the individual workstations on the factory floor. To perform these tasks, the central computer needs a general-purpose, widely used, widely supported OS such as Unix.

To understand how Unix serves both the real-time world and the general-purpose world, it is necessary to understand how real-time systems differ from time-sharing systems. Two common multitasking system features—process scheduling and process blocking—can illustrate how Unix differs from a real-time system.

The Scheduler

Multitasking operating systems include a *scheduler*, which has the responsibility of allocating CPU time among the various loaded processes. Even though both Unix and most real-time operating systems have schedulers, their basic algorithms differ sharply. A real-time system is *event-driven*, whereby the OS allocates CPU time to processes that need to service events. (*Events* include new input data, a clock time-out, and so on.) The scheduler in Unix, on the other hand, is a time slicer; it attempts to give all processes a chance to run.

The system (or the user of the system) may assign a priority to each process in the system. For a real-time system, a high-priority process with outstanding events will have near-absolute control of the CPU; only a hardware interrupt can intrude on the process. A low-priority process will run only when the high-priority process can no longer run (for example, when it must wait on some new external event).

Under Unix, the OS may give a high-priority process a larger slice of CPU time than it does a low-priority process. However, when the high-priority process has used up its time slice, the OS suspends it and begins execution of another process.

Asynchronous and Synchronous System Requests

At any given time, a process is in one of three broad states: *running*, in which the process has control of the CPU; *runable*, in which the process would like to run but

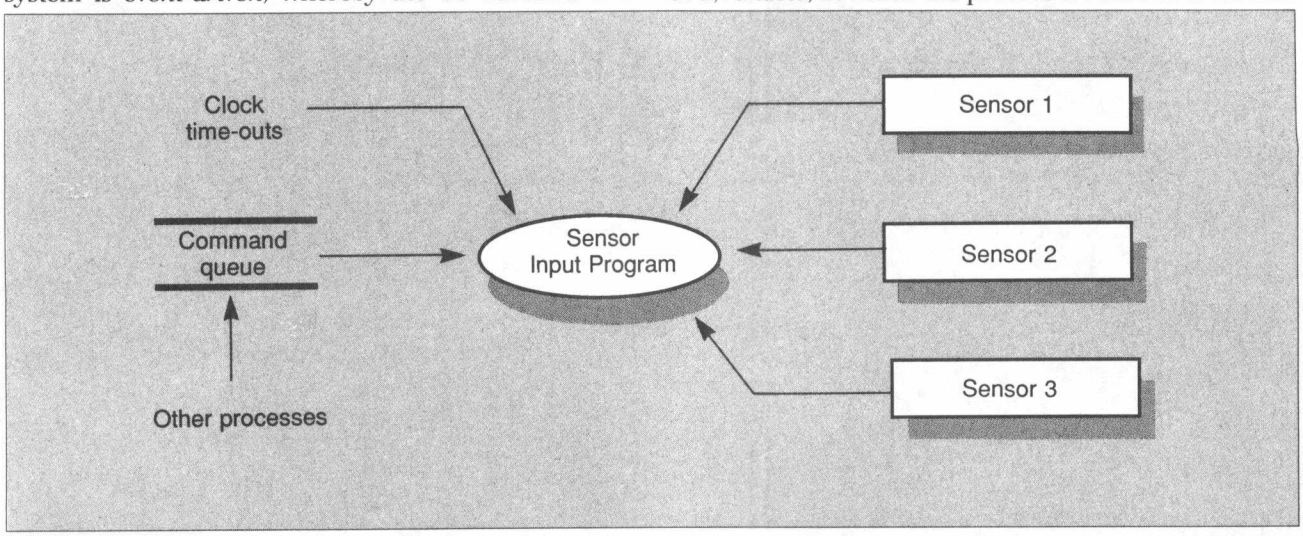


Figure 1: *Data flow for solution A*

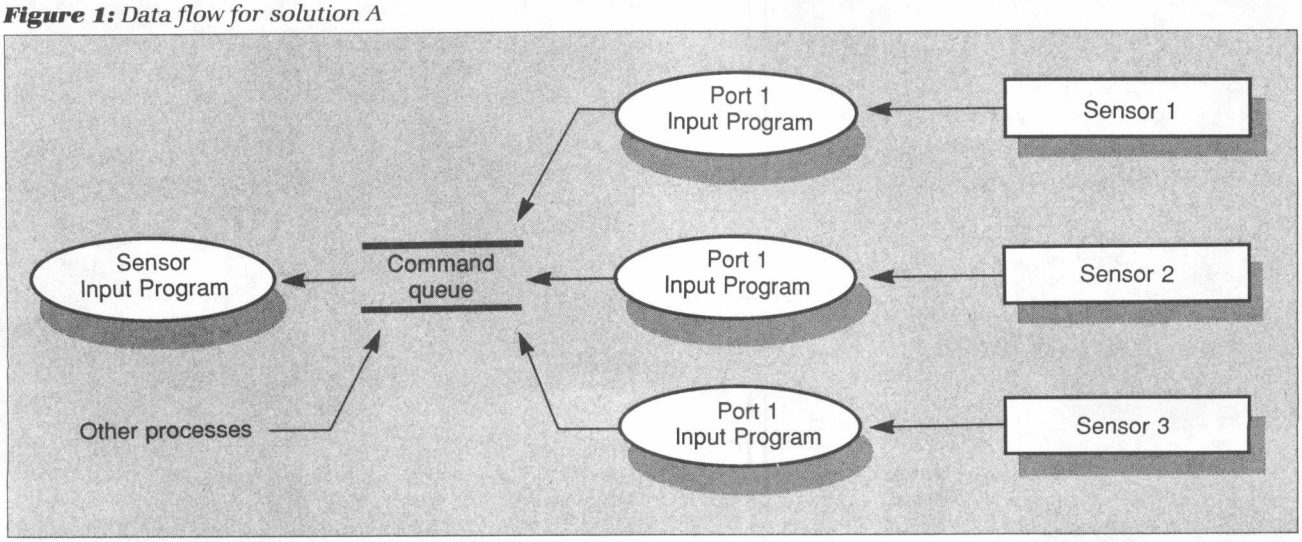


Figure 2: *Data flow for solution B*

(continued from page 20)

some other process currently has control of the CPU; and *blocked*, in which the process is waiting on some event—a clock interval, operator input, output to flush, and so on.

Unix, like many other systems, makes most of its OS requests *synchronously*, and they become blocked until the system can service them. Most real-time systems, on the other hand, make systems calls that may lead to blocking *asynchronously*, whereby the calls make their request for the resource but continue normal processing. When the resource becomes available, the OS notifies the process in one of two ways—either with an *event flag* or through a *completion routine*. Many real-time processes use a combination of event flags and completion routines.

Event Flags

Event flags are semaphores that indicate whether or not a condition has become true—for example, a process may ask the system to read a device and set the event flag when it has finished reading in the data. A real-time process may have several outstanding asynchronous system requests pending—the program may make a request to read input from a port, it may schedule a clock time-out, and it may request a read from an interprocess queue. Real-time operating systems allow the process to become blocked until one flag or a combination of flags become set.

Unix has a similar concept, called *semaphores*. The most common use of semaphores, however, is to provide locking on some shared resource such as a shared memory table. Unix provides no way to link a *read( )* request to a semaphore, for example; hence it provides no way for a process to become blocked while waiting for the system to service multiple requests.

Completion Routines

Completion routines are called by the OS on the behalf of a process when the OS has finished with some requested function. (In some environments completion routines are also known as *asynchronous system traps*, or *asynchronous system routines*.) Generally, you can think of completion routines as the software equivalent of hardware interrupts.

If the OS finds that another event has become true while the process is still handling the original completion routine, it can either handle that event right away or else queue up the routine and handle it after finishing with the current completion routine. Some real-time systems also allow prioritized event handling, like CPUs that prioritize hardware interrupts.

Most systems allow a program to pass some predefined data to a completion routine; hence the program may have a single completion routine servicing similar requests. You may want to set up read requests on several sensors using the same completion routine for each; when the system asynchronously calls the completion routine, it will pass some argument block identifying the particular sensor that has been read.

Unix has no concept of completion routines, and because all resource requests are synchronous, it has no need to notify a process that a particular request has been completed, either through an event flag or by calling a completion routine. (Unix allows you to input calls that return immediately if no data is present. Although this doesn't provide an asynchronous interrupting ability, it does allow a program to set up its own polling loop without becoming blocked waiting on a single event. One of the examples I discuss later uses this sort of "no wait" input.)

Unix does, however, have a construct called a *signal*, which behaves similarly to a completion routine in that when a process receives a signal (from the operating system or from another process), the signal will begin execution of some predefined routine. Signals were designed to handle asynchronous hardware exceptions such as bus, floating-point, and addressing errors as well as user abort requests and program time-outs.

Once a process has intercepted a signal, the expected response is usually to perform some cleanup (close files, release resources, maybe print an error message) and then exit. By default Unix assigns a set of routines to handle this cleanup; however, a program also has the option of setting up its own functions that Unix will call when it receives a particular signal.

The signal-handler function can perform any operation. Because of the nature of the operating system, however, while in the handler code, the process is in a very fragile state. For example, after Unix calls the signal-catcher routine, it automatically resets the signal catcher to its own default value. If the process receives a second signal before the handler has had a chance to reset the signal catcher, Unix will call its own routine, which may abort the program.

The only signal type that you can use successfully as part of your normal programming is the clock time-out signal, *SIGALRM*, which is a special type of signal that you can schedule from within your program via the *alarm( )* system call. Although you cannot predict exactly when your program will receive the *SIGALRM* signal, you can keep track of whether or not your program has issued the *alarm( )* call; hence your program can take some precautions to handle the interruption. One of the sample programs accompanying this article illustrates a "safe" *SIGALRM* handler.

Two Alternatives

When building a real-time Unix process or porting an existing process from a real-time environment, you can choose two basic plans of attack. The first involves simulating the real-time environment by setting up a scheduler routine within your process. Although your scheduler won't be as time efficient as a real-time OS scheduler, this method does have the advantage that the Unix version will have the same basic structure as the real-time version. This is particularly attractive when porting an existing process to Unix.

The second method of building a real-time process requires that you understand Unix's limitations in the real-time environment and structure your process so that it takes advantage of Unix's strengths. As mentioned

(continued from page 24)

earlier, Unix has no real notion of asynchronous system resource requests nor of the real-time constructs used for implementing them (event flags and completion routines). The key, therefore, is to write your processes so that they don't require asynchronous system requests.

That last statement seems intuitively obvious and may seem intuitively impossible as well. With a little forethought, however, you can create a process that will run with nearly the same efficiency as a version running on a dedicated real-time operating system.

The Problem

Suppose, for example, that you have a process whose primary duty is to read a set of sensors and process the data. If the process doesn't receive any input from the sensors after some time interval, it should alert some other process. In the background it may also need to deal with commands coming from other processes. Figure 1, page 20, shows the basic data flows.

For the sake of simplicity, let's assume that the sensors are connected via a standard serial port and that sensor data comes in over the link in new-line-terminated ASCII strings. The algorithm and code omit the actual input processing as this isn't important to the example. The program listings also omit error checking; for any production program, you will, of course, need a generous portion of error checking.

Solution A—Overriding the Unix Scheduler

One method for solving the problem is to make Unix believe that it is a real-time OS. This solution is inferior to Solution B (described later), but for some applications, particularly for quick-and-dirty ports from real-time systems, you may prefer it.

Listings One–Three, pages 50–55, show the solution written in C. Notice that the main loop attempts to read each of the sensors as well as the command queue. If *read( )* finds data present, it will process the data; otherwise, it will continue polling the other inputs. To prevent the process from hogging the CPU in an endless loop, it will delay between loop iterations.

This program includes two functions—*nap( )* and *marktime( )*—that are worthy of further discussion. *Nap( )* provides a program delay of a finer granularity than Unix normally provides with the *sleep( )* system call. *Marktime( )* provides a consistant interface to the *SIGALRM* clock signal.

Nap( )

The Unix system call *sleep( )* delays a program with a granularity of 1 second. Most versions of Unix actually implement this by delaying the program until the next second boundary after the indicated time rather than an exact number of seconds following the point at which your program calls *sleep( )*. Hence, *sleep(1)* may delay your process anywhere from a single clock tick to nearly a full second after invocation. In most instances, this is acceptable. For the sample program, however,

let's arbitrarily decide that you need to poll the input sensors every 300 milliseconds by using the *nap( )* function as shown in Listing Two.

Nap( ) works by using a special characteristic of the standard terminal driver. Normally, a terminal driver reads characters from a port until it sees a carriage return and then returns the input string to the process. This is known as *canonical processing*. Canonical processing also understands user-defined ASCII characters for backspace and line erase. Canonical processing is appropriate for reading input when the user is typing in input from a terminal.

Via an *ioctl( )* system call, you can disable canonical processing and set two input parameters—*VMIN* and *VTIME*—so that a *read( )* from a port will return after either reading *VMIN* characters or else delaying *VTIME* ticks (expressed as tenths of a second). *Nap( )* uses (some may say abuses) this characteristic by attempting to read from an unused port. Naturally, it will find no characters available and will return a failure after a delay of *VTIME*.

Marktime( )

I mentioned earlier that the Unix signal provides a function similar to a completion routine. The *alarm( )* system function call allows a program to schedule the signal *SIGARLM* sometime in the future, and the *signal( )* system function call allows the program to define a signal handler. By themselves, *alarm( )* and *signal( )* provide a limited means of implementing a completion routine. They are limited in that only one alarm can be outstanding at any given time and that they have no implicit means of passing data to the completion routine. Also, a program that needs to set alarms from different parts of the program has no means of coordinating the different time-outs except through the use of global data—a poor programming practice.

Marktime( ) and its associated functions form a shell around *alarm( )* and *signal( )*. They allow the calling program to set up multiple time-outs; each time-out can have its own completion routine, a flag that will be set when the time-out expires, and a pointer to a unique data block. The calling process can also disable, enable, and cancel time-outs through the functions associated with *marktime( )*. Note in Listing One that the program disables time-outs when not explicitly processing the main loop; this ensures that the time-out won't unexpectedly interrupt other processing.

Listing Three shows *marktime( )* and its associated functions.

Solution B—Restructuring the Problem

For most situations, the preferred method of solving the problem is to work within the limits of Unix. Remember that Unix processes can only block on one system call at a time, typically a *read( )*, a *pause( )*, or a *sleep( )*.

Instead of trying to make a single Unix process handle all the possible events, as I did in solution A, you can reorganize the main routine into several small proc-

esses. Each of the small processes will be responsible for a single blockable resource; when one of the processes has completed its duty (for example, when a sensor input process has read a block of data), it will pass a message back to the main process via an IPC message queue. The main process will then have only one blocking state, which is a *msgrcv( )* of its message queue. Figure 2, page 20, shows the data flows for this solution.

Although this method incurs a degree of overhead in the form of message passing, it simplifies the problem by removing the polling/sleeping loop, and hence the overall performance will generally be better than that in solution A.

Listings Four–Six, pages 61–65, show the C code for this solution. Note that the listings don't show how the system starts the smaller processes; in a real application the main process may *fork( )* and *exec( )* the smaller processes as children. Also, note that the child processes have no means of relating exception conditions to the parent process; one way to do this would be to define additional messages.

Finally, note that the child processes break the *read( )* using a *marktime( )* time-out. An alternative to this would be to use noncanonical input from the port using the *VTIME* and *VMIN* parameters to control the time-out. The down side to using this method is that the terminal driver no longer parses according to new-line delimeters, which means that the program must handle its own parsing of the input stream. Although this method may provide a cleaner interface, particularly if the input is a binary stream rather than ASCII, I used *marktime( )* to illustrate its use with a pseudoevent flag.

Conclusion

Both solutions have merit under the right circumstances, and both can successfully perform the required chores. The best solution to running a near-real-time process under Unix, however, lies in understanding the nature of the operating system—its strengths and weaknesses—and working with the system instead of against it.

Although the solution requires some extra nontraditional planning (nontraditional in the sense of how you might build the product under a real-time operating system), restructuring the problem as I did in solution B provides the most efficient and trouble-free product.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 50.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

MESSAGE-PASSING OPERATING SYSTEMS

High performance and flexibility in a realtime, multitasking, multiuser, or networked environment.

by Dan Hildebrand

Operating systems in real-time environments must be capable of handling the demanding intertask communications problems that are inherent in communications, process control, and other real-time applications. At the same time, the operating system (OS) must be capable of providing the functional capabilities of a traditional multiuser OS while delivering fully network-distributed processing and the deterministic performance of a real-time executive. One way to achieve both performance and flexibility is message passing architecture.

The performance and flexibility of a message-passing OS enable the data flow on the network to consist of intertask messages. Tasks can communicate with any other task, anywhere on the network. The network then functions as a homogeneous, tightly connected array of computers, rather than a collection of computing islands connected on a functionally limited network.

Dan Hildebrand is a programmer for Quantum Software Systems Ltd. (Ottawa, Ontario) and the developer of Qterm, a high-level terminal emulation program.

Conventional wisdom would have us believe that the 8088 and 80286 posess a "flawed" architecture, leaving them unsuitable for multitasking. Contrary to popular opinion, the design of these processors are admirably suited to multitasking. It is only the "flawed" architecture of conventional operating systems that negatively impact their performance. Operating system design is the primary limiting factor in the multitasking performance of these processors.

The Dilemma of the Layered Approach

In a conventional OS, various unrelated pieces of the OS often share common code and data space for convenience of implementation. Software layers over existing facilities (the "yet-another-layer" design philosophy) provide additional OS function ability. With each new release, this ever-increasing depth of layering results in progressively worsening performance in the following three crucial areas:

- The synchronization overhead incurred when a task communicates a request to the operating system.
- The transparency of intertask communications.
- The efficiency of intertask communications.

For example, to provide network services, a layer is often added to catch OS requests and reroute them through the network software/hardware to a file server. The file server is then running a network control task that interfaces the network requests to the local OS. This "network-services" layering imposes a performance penalty on all network transactions.

To avoid performance losses, extensions can be coded into the kernel itself, thereby having access to data structures and code fragments not necessarily needed for the extension. However, these pathological connections result in side effects that can be difficult to debug and maintain. You face the dilemma of choosing between:

1) extending the complexity of the OS kernel at the expense of reliability and maintainability; or

2) extending the OS services through the addition of multiple, performance-robbing layers around the existing OS.

SideStepping the Dilemma—The Message Passing Solution

One technique that solves the performance difficulties of intertask communication is message passing. This technique involves the copying of a block of data (the message) by the OS kernel from the data space of one task to that of another. Whether the tasks are executing on the same processor or on physically remote processors does not matter. Obviously, this approach is particularly effective in integrated-network, distributed, and parallel processing environments.

An important characteristic of this approach is that message data must be physically copied from the source task to the destination task. This physical copying of the message accomplishes a "disconnection" between the two tasks, thus allowing the tasks to run on different processors (if necessary). If one of the two tasks provides OS-related services, this disconnection easily results in a networked operating system.

While performance optimization techniques may encourage the passing of pointers to messages (rather than passing message contents), this optimization has negative effects. In actual practice, the time for data transfer (passing the message) does not represent a significant portion of the task-switching process. The task switch itself represents the bulk of the operation. Also, the vast majority of messages are only a few bytes long.

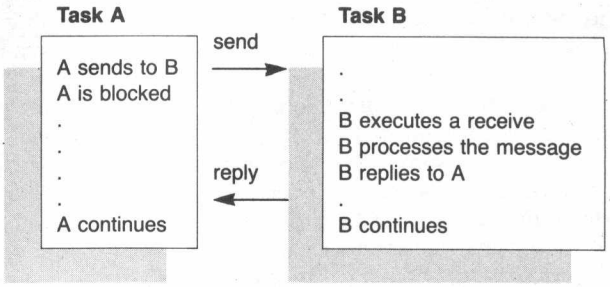
In specialized applications where large buffers must be passed between tasks and where networking is not an issue, a pointer to the necessary buffer can be passed within the message. If the message is not physically copied, many additional details must be managed. The primary problem is that a sending task cannot modify or release a message buffer until the receiving task has indicated that it is finished with the message. The synchronization issues that must then be addressed only complicate and impede the operation of the system.

Conventional, layered operating systems typically protect themselves from user tasks by rigidly separating memory into "system" and "user" areas. An OS built from a group of cooperating tasks that pass messages can be set up without distinct system and user memory spaces. The only necessary system memory management is that already provided to support user tasks. A system task is then treated the same as a user task except that the system task is providing a resource intrinsic to the OS.

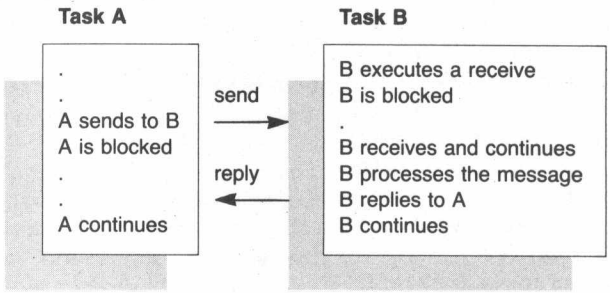
The dilemma confronted when expanding a conventionally structured OS is neatly sidestepped with this multiple task arrangement. Extensions to the OS are painlessly added as additional tasks that efficiently pass messages to the existing OS tasks. Maintenance of the OS also can be easily managed because each task is responsible for only a well defined set of services, requested through an explicitly defined set of messages.

If memory-management hardware is available, an additional benefit of this structure is that all user and OS tasks are protected from one another. The 80286 microprocessor running in protected mode is an example of an environment within which this protection is available. The modular nature of this type of OS design is highly reliable and easily

There are two sequences that can occur when two tasks are communicating with each other using the send/receive operators. If Task A sends to Task B before Task B is ready for a receive, the following sequence occurs:



If Task B executes a receive before Task A does the send, the synchronization operates as follows:



Throughout this process there are a number of states a task can assume. These states are "send blocked," "receive blocked," and "reply blocked."

Send Blocked
The task has done a send which has not yet been received.

Receive Blocked
The task has done a receive before any other task has sent to it.

Reply Blocked
The task has done a send which was received but a reply has not yet been sent back.

Figure 1: *Send/Receive Message Passing*

modified and maintained.

Message-Passing Operating System Interface

The application interface to a message-passing OS is quite different from the OS interface provided by operating systems such as OS/2, PC-DOS, or Unix. Such operating systems require the application program to execute a software interrupt, or subroutine calls passing either the data for the request in the processor registers or through a pointer to a predefined table or buffer. The OS then expects to be able to directly read and write into the data space of the application making the OS request. This requires that the OS be resident on the same CPU as the task making the request and results in severe problems for networked versions of these operating systems.

Additional layers of software (which decrease overall system performance) are required to solve this problem. In contrast, a message-passing interface produces an OS with a single, unified interface that works for communication between either local tasks or remote tasks. This unified interface results in a smaller, leaner OS that need not support two sets of interfaces. While other OS interfaces require the applications to be written in both "networked" and "non-networked" flavors, as well as requiring the operating system to support two sets of interfaces, a message-passing OS means that an application in a message-passing OS need only be written for a single interface.

One such message-passing operating system is QNX, designed and developed by Gordon Bell and Dan Dodge as an outgrowth of research done at the University of Waterloo in Canada. This operating system was introduced in 1982 by Quantum Software Systems Ltd. It is currently being used at over 55,000 sites in applications ranging from integrated office automation systems to robotics and real-time process control systems.

Although its underlying architecture is much different from Unix, the QNX interface itself is Unix-like. The OS consists of a group of cooperating tasks that pass messages among themselves in order to accomplish various OS requests. These tasks are referred to as administrator tasks because they are essential to the operation of the OS. When an application task requires OS services (such as device I/O, task creation, and so forth), messages are sent to the administrator task that provides the required service. If those services are required of another workstation or node in the network, those same request messages need only be sent to the administrator tasks on the remote node. This message redirection is handled transparently by the system.

The kernel holds together all of the administrator tasks. The QNX kernel, which represents 10K of highly optimized code, has the primary function of performing the message-passing and task synchronization functions within the OS. A task scheduler that has set priorities within the kernel provides QNX with the deterministic response time necessary for real-time applications. On an 8-MHz 80286, the kernel performs 3,200 task switches per second; on a 16-MHz 80386, 7,200 task switches per second. Assuming another interrupt is not being serviced, a worst-case interrupt latency of 30 microseconds is experienced on an 8-MHz 80286.

System Administrator Tasks

Various administrator *tasks* are placed around the kernel. "Task" is the task which provides facilities relating to task creation, task death, memory allocation. and task name registration. These are given the highest priority in the system. The protected-mode 80286 version of QNX supports 150 tasks, while the real-mode 80286 or 8088 versions support 64 tasks.

Fsys is the task that implements the QNX file system. It manages the on-disk data structures that represent files and directories. Messages can be sent to this task to request operations related to the file system (such as file opening, closing, reading and writing, as well as absolute disk block manipulation). Fsys implements a tree-structured file system that supports disks up to 1 Terabyte (a million Mbyte) in size with a space-efficient 512-byte unit of allocation. This file system supports random seeks within files from any point to any other point with a single, direct disk seek. Unlike typical file systems, intervening disk blocks need not be read to perform large seeks. This means QNX can be used for large, multiuser database applications. Because the file system is also power-fail safe, QNX is also suitable for harsh environments. File ownership, attribute and permission checking usually found within a multiuser file system is also handled by Fsys. Block-oriented device drivers can be installed using a "mount" command and become an extension of the Fsys task. See accompanying text about "Mounting Device Drivers." Special tasks can also be written to adopt a drive for special purposes.

Dev is the task that performs character-oriented I/O. Drivers for the console, serial, and parallel devices are present within this task. Additional drivers can be mounted as background tasks, which can then adopt device names from the Dev task for special applications. The drivers within this task perform all the handling for options (such as flow control, line editing, baud rate changes, and so forth). Changes to option settings are performed by utilities that send the appropriate messages to Dev, thus commanding Dev to modify the requested options. Also present are library routines that allow user programs to communicate these requests. A set of routines that implement high-speed video output are included in Dev. Since these routines are integrated into the terminal independent screen and keyboard library, programs can be written that perform instaneous screen updates on the console, while retaining terminal independence for terminal or modem applications. On a PC AT, 19 physical devices are supported, in addition to the 40 virtual device names available for adoption by device-driver tasks. The fast task switching and low interrupt latency of QNX allow many more serial devices to be supported than under conventional, non-real-time,

Unix derived operating systems.

Idle is a null task executed whenever all the other tasks in the system are in a blocked state and waiting for an external event either to occur or to complete. Idle runs at the lowest priority in the system.

Net is the task within QNX that performs message passing between machines on a network. This task exists only in networked versions of QNX and occupies approximately 20K of memory. (By comparison, the standard networking extension for PC-DOS is nearly 190K in size.)

The user-extendable *Timer* is an optional task that can be started by the user to add complex timing capabilities. Other tasks can request "timer" to provide timeouts ranging from one millisecond to many years. Because of the real-time scheduling within QNX, tasks can be accurately scheduled with very precise timing resolution.

The *Queue* manager is a task that can perform queued message passing similar to that provided in mailbox-oriented operating systems. The standard send/receive, intertask message calls within QNX are blocking. Unless a conditional receive has been explicitly requested, these calls do not allow the sending task to continue until the message has been received. This blocking design is deliberate within the operating system, although it may not be convenient for some system designs. To support those designs that require it, the Queue manager task can be started to provide network-wide, queued message passing. Unlike the OS/2 Queue manager, this Queue manager buffers entire messages, rather than just pointers to messages. This allows queues to be used across the network (if necessary). If

OS/2's A Real-time Alternative

Neither Microsoft nor IBM touts OS/2 as a real-time operating system. Nevertheless, programmers might write OS/2 applications that must track real time. This is particularly true when programmers are developing communications applications to monitor events and take action when responses fail to occur as expected. Real-time tracking can provide the user with a specific time period in which to perform some action, or perform an action such as saving an editor's buffer on a regular, timed interval. Real-time control can also allow an application to run at preset time intervals. OS/2 has several timer service functions to facilitate writing real-time control routines.

The only fly in the OS/2 real-time control ointment is OS/2's main feature—multitasking. Because multiple threads and processes can be running simultaneously, real-time tracking that uses the CPU clock can never be totally accurate because a higher-priority process may be eating up CPU cycles. But multitasking has its advantages, too. Using multiple threads, a program can synchronize several different hardware devices so that they can perform simultaneous tasks. Timers can likewise synchronize the activities of several asynchronous programs.

OS/2 provides both synchronous and asynchronous timer services. DosSleep is the synchronous function that puts your application on hold so that you do not need delay loops. DosTimerAsync, DosTimerStart, and DosTimerStop are asynchronous functions that allow you to start, stop, and read software timers, using system semaphores to alert an application when timing functions have finished executing. The timer starts when it is called and then control passes back to the calling thread, which resumes execution. The thread and the timer execute concurrently. Upon completion of the timer's interval, the timer clears a semaphore. The calling thread can check the semaphore to see if timing is complete. For example, if your program issues a DosTimerAsynch(5000, mysem, semidentifier) call, a timer with a five-second interval begins execution and at the end of the interval clears the semaphore mysem, which you can read on the file handle semidentifier. Your program must create and set the semaphore by using the DosCreateSem and DosSemSet functions before you call the asynchronous timer.

DosTimerStart operates much the same as DosTimerAsynch but continues to run while clearing its associated semaphore each time the timer interval elapses. You must reset the semaphore after it is cleared. DosTimerStop is used to halt DosTimerStart. The DosSleep function acts as a synchronous timer. The thread that calls DosSleep suspends its execution for the interval of the timer. A DosSleep(5000) call puts its calling thread on hold for five seconds.

If your programs merely need to synchronize the flow of data among threads or processes, semaphores and shared memory enable you to exploit one of several OS/2 communication paths between processes. Particularly within a single monolithic application, the private semaphore/shared memory interprocess communication technique has speed advantages over the nonprivate (but slower) pipes mechanism for sharing data. Pipes pass data only between parent processes and their children. Two-way communication between such processes requires two separate pipes.

The periodic clock interrupt (or timer tick) of OS/2 occurs 32 times each second. This means that timing functions carry a 1/32-second quantization error. Therefore, you should think in term of seconds (not milliseconds) when using timers of OS/2. High-precision timing of events happening in the millisecond range should use other methods to measure time intervals. For example, you can make repeated DosGetDateTime calls to read the Date/Time date structure's contents into a buffer or into variables for computing the passage of small time intervals. Another solution is to check

performance optimization is necessary and network transparency is not important, the message stored in the queues can be a pointer to the message.

Through a request to Task, user-written tasks are able to become admin tasks. As admin tasks, they share the same privileges as the original set of tasks that make up the OS. Being able to start admin tasks allows the initial functional capabilities of the OS to be extended at run-time.

An essential characteristic of an admin task is that it cannot be arbitrarily killed by other tasks. Typically, admin tasks are commanded to shut down and to release any system resources the tasks may have allocated. An admin task is also able to detect the death of other system tasks so that resources allocated by the admin task for those dead tasks may be released.

Send/Receive Message Passing Primitives

QNX implements two message passing primitives—send and receive. These primitives are unbuffered, blocking operations that cause the task issuing a send request to be blocked if the target task is not correspondingly receive-blocked. When two tasks are in complementary send/receive states, the message is transferred and the receive task becomes unblocked (see Figure 1, page 35). The highest-priority task will then run. QNX always executes the highest priority, unblocked task. If two tasks are compute bound at the same priority level, round robin task scheduling will occur. Should an event occur that causes a higher priority task to become unblocked, QNX will pre-empt the currently executing task, and switch to the higher priority task. Fast, pre-emptive task scheduling is essential to real-time applications.

An important aspect of the send/receive operations is that time-ordered queuing is performed whenever more than one task attempts to communicate with the target task. Multiple send requests to a single task performing a receive are queued in the order they were received and are processed in sequence. The target task has the option of completely servicing the first request before serving the next request (or additional requests).

In addition to the send/receive operators, mechanisms called exceptions, ports, and registered names are available for intertask communication. These additional mechanisms are useful for special cases where send/receive communication is not appropriate.

An exception is similar to the signal found in Unix. An exception is an asynchronous event that can cause an exception handler within the task to be executed in response to the exception. Exceptions are valuable because they can be used to break out of the send/receive blocked states. The most common exception

the time bytes in the read-only global information segment by using the DosGetInfoSeg call.

You can alleviate the problem of timer service threads losing clock cycles to higher priority threads by elevating the priority of the thread making the timer calls. If you make these threads the highest-priority threads, you will ensure that events needing critical timer servicing won't lose clocks to higher-priority threads. The major difficulty developers face in writing real-time control softare is interrupt processing. OS/2 does not allow an application to process hardware or software interrupts. You can process interrupts only with OS/2 device driver. Since these drivers are very difficult to write in a high-level language, creating routines to handle interrupts—which can then notify and pass data to an application—will be a complex process. In short, OS/2 should be considered a workable real-time operating system alternative only for those custom applications where you are in total control of the environment and can therefore, worst case interrupt latency and task scheduling times. —G.M.V.

G. Michael Vose, co-editor of the newsletter "OS Report: News and Views on OS/2." He can be reached at Box 3160, Peterborough, NH 03458.

Mounting QNX Device Drivers

Within QNX, device drivers can be started or mounted at run time and do not need to patch nor be compiled into the kernel of the operating system. Drivers for block-oriented devices (hard disks, floppy disks, RAM disks, and so on) can be written as specially structured interrupt handlers which are connected as particular drive numbers (using the mount command) to the Fsys (file system) task. This type of driver becomes a directly callable set of routines within the code space of the Fsys task which are able to do low-level block read/write operations while the Fsys task manages the file system data structures within those *blocks*.

Mounted Fsys drivers are automatically used by the Fsys task when access to the appropriately numbered drive is requested. Since the Fsys task is network accessible like any other task, the drivers attached to Fsys are accessible using standard file system calls (*fopen*, *fclose*, and so forth).

A mechanism to adopt a disk drive also exists, allowing any foreign disk format to be easily supported with a driver task for that format. From an application's point of view, when the *fopen* message for a given drive is sent to the Fsys task that owns the drive, a special reply message will be returned, indicating which task has adopted the drive. The *fopen* library routine will then resend the same *fopen* message to the task on the node that has adopted the drive. From this point on, all further requests for that drive will be routed directly to the task that has adopted the drive.

This facility is used to advantage by the DFS task (Dos File System). When started, this task can bring into the device list any PC-DOS hard disks or floppy disks, anywhere in the network. Once mounted, any QNX program that accesses files in the QNX file system can also access any PC-DOS file system and have the file structure information translated "on the fly" for any request. For example, with the DFS task running, the QNX full screen editor can edit any file on any disk (QNX or PC-DOS) without regard for the underlying structure of the file system.

Using this technique, it is possible to write a file system administrator for a WORM drive. This drive could then be read and written by any of the standard QNX utilities. It would be up to the WORM file system task to maintain a file system on the optical drive that could handle the write-once limitation of the drive.

Drivers for character-oriented devices can be written as a task that adopts a new or existing device name so that additional devices can be accessed like any other serial or parallel device using the standard file/device calls (*fopen*, *fclose*, and so on). The implementation of a spool device which collects output in a disk file rather than printing it is an example of such a driver.

Device drivers can also be written in the form of background tasks that implement their own message-passing interface. Complex devices such as MAP (Manufacturers Automation Protocol) interface cards do not naturally interface as block or character-oriented devices and should be implemented as a standalone control task. The network wide intertask messaging of the network then makes the service provided by the driver task a network-accessible resource.

Writing a driver for a network card that is connected to a network separate from the QNX network produces a gateway. The machine on the QNX network that contains both network cards would then run the gateway task. This gateway task could then register its name on the network, becoming a network accessible resource that can allow any QNX task throughout a network to access the remote network via the gateway task. —D.H.

MESSAGE-PASSING

is the break exception that is generated from the keyboard.

The primary use of a port is for interrupt handlers to communicate with a task. This facility makes it possible for a task to be written which contains the interrupt handler within the body of the task itself. Using standard systems calls, the task is able to attach to a port and then connect the handler to the appropriate interrupt vector. After any other internal initialization, the task can receive block itself upon the port and optionally open itself for message reception from other tasks. An "attach" or "detach" operator is used by a task to obtain a port.

During interrupt service time, the interrupt handler is able to make use of the code and data of the task. If an event requiring handling by the task or the OS results, the handler can signal the assigned port, causing the task to become unblocked in order to perform whatever service the interrupt handler required. Note that the interrupt handler is connected directly to the interrupt vector and that no operating system overhead is added to the interrupt service time.

For two tasks to communicate, the sending task must know the node number and task identifier (ID) of the destination task. If the sending task was responsible for starting the remote task, the sending task will know this information. If the sending task is expecting to send to a previously present task, the sending task must be able to discover the node and task ID (TID) of that task. To facilitate this, the receiving task that wishes to provide a network-accessible service can register a textual name. Tasks needing to locate that task can obtain the node and TID by using the textual name that the task would have registered.

For example, one task that typically needs to register itself is a print spooler. Once registered, any task on the network wanting to print need only inquire about the spooler task and use the standard intertask messaging to send the data to be

printed to the spooler task. Multiple spoolers could be started for any printer, anywhere in the network. Any spooler and its corresponding printer could be relocated without concern for whether tasks needing that spooler would be able to locate it.

Since the primary tasks within the operating system are assigned predefined TID numbers at boot time, remote tasks are always able to communicate directly with the primary OS tasks anywhere on the network without having to first discover their TID numbers.

Flexibility

Operating system extensions to QNX are background tasks. These extensions interface to application tasks with exactly the same messaging that applications already use to request OS services. In effect, user-started OS extensions become indistinguishable from the OS itself. These additional tasks run under the privilege and access restrictions necessary for their intended purposes and no more. This allows the protection inherent in a protected-mode OS to also apply to user-written OS extensions.

The segmentation of the OS into a set of cooperating tasks is also memory-efficient. Portions of the OS that are not needed for every application environment can be implemented as background tasks that are started only as needed for a given application environment. For example, the QNX network administrator task automatically removes itself from the system if it detects that a network card is not present.

A further example of the flexibility of this approach is that QNX is able to support PC-DOS as a task within both the real-mode or protected-mode versions of QNX. Most PC-DOS applications (including Lotus, Sidekick, dBase, and WordPerfect) can run without difficulty.

Networking and Message-Passing Operating Systems

Message-passing operating systems provide an alternative approach to networking. Most networked operating systems deal with file transfers or remote terminal-session communications. The data flowing on a message-passing network like QNX, is the intertask messaging itself. Operations such as command loading (file transfer), remote device I/O (terminal sessions), and general intertask communications are transparently accomplished within the context of intertask messaging.

User tasks are able to make requests of any task in the system, anywhere on the network. As a result, the network provides each task with what appears to be the resources of a parallel computer, with all network resources (processors, memory, disks, devices, and tasks) accessible as local resources.

Rather than just a simple copy of a file to a device, as with the command "copy file $lpt," QNX allows commands to be specified in a manner that takes full advantage of all of the available network resources. The next copy command demonstrates this:

MESSAGE-PASSING

```
$ [2] [3]4:/cmds/copy [1]3:/user/peggy/
                              new [4]$lpt
```

Assuming this command line were typed from the console on node 6, the command copy would be loaded from node 3, drive 4 into memory on node 2. While executing on node 2, the copy command would read data from the file new on node 1, drive 3, and send it out to the $lpt device on node 4. If a '&' character were placed on the end of the command line, the command would execute as a background task on node 2 and freeing the user's console for other work. None of the general utilities contain special code to handle network-oriented file or device names.

In this case, the "copy" command itself is 2,700 bytes in size and operates with simple calles to standard file procesing library calls.

Device access across the network is able to work in this manner because whenever a task performs an *fopen* to open a file or device, it sends a message to the fsys (for files) or dev task (for devices) on the node that owns the file or device. When the requested task replies to the *fopen* request, any following read/write messages will be routed directly to the remote task that controls the device.

In a Unix environment, the network typically supports only terminal sessions and file transfers. To access a remote database, the task performing the I/O must log in to the remote node and execute the query task there. The result is the central machine (already burdened with the file and device I/O) must also execute the tasks that are generating the I/O requests for all the users on the network. A QNX network allows the tasks to execute on each user's workstation with only the remote file and device I/O requests flowing to the node (or nodes) that contain the devices being accessed.

With this naming flexibility, resources present on the entire network are part of the same "name space" and may be operated upon just as the resources present on a single node. A program written to access files or devices by name can name any file or device on the network. The program can then have transparent access to the file or device without resorting to a special "network services" interface.

Conclusion

Through the use of a message passing architecture, QNX is able to provide real-time performance with multiuser support and full network transparency in an operating system that occupies less than 150 Kbytes, QNX can run on as limited a machine as a PC with a single floppy drive and 256K of RAM, or in protected mode on a 80286 or 80386 with 16 Mbytes of RAM. The networking transparency results in a QNX "mainframe" that can be built piece by piece to provide as much performance as necessary.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

A Simple Decompiler

Recreating source code without token resistance.

by William May

On the surface a decompiler sounds like a hackers' tool, a utility for snooping through code. Although programs such as Steve Jasik's MacNosy for the Macintosh are certainly useful, the decompiler I describe in this article has more prosaic, less imaginative uses. In my case its purpose was to speed up a product that included a spreadsheetlike component.

Fast recalculation speed is an important objective of a spreadsheet, and I felt this would best be achieved by converting the user's expressions into a form that could be computed efficiently—that is, a compiled form. Although the compilation process is straightforward, a problem arises when the user wants to see and/or change an expression: The application must be able to recreate the original expression, or "source code."

A simple way to satisfy this need would be to store both the source and the compiled expressions. This method is used in Smalltalk-80, which also provides interactive viewing and modification of compiled expressions. I felt, however, that with the limited syntax of spreadsheet expressions, I should be able to recreate the source code from the compiled code itself.

The problem turned out to have been solved already, and the algorithm was described in an article by P.J. Brown.[1] As you will note from the publication date (1976), the algorithm was devised before the introduction of spreadsheets or of microcomputers. Brown developed the algorithm as a component of an interpreted Basic system for minicomputers, where it was used to decompile Basic statements into source form. Given its heritage, it is clear that the algorithm is capable of handling a far wider range of expressions than my spreadsheet is.

The algorithm has a few points that should make it of interest to *DDJ* readers. First, as I said, it can handle an extremely wide range of expressions, which allows it to find use in many applications. Second, the algorithm is driven by a table that is easy to set up and maintain. With its flexibility and simple setup, the algorithm is one of those tools that, once available, seems to find many unexpected uses. Finally, the subject of decompilers is worth exploring.

The remainder of this article covers my implementation of Brown's algorithm. The code, developed on a Macintosh in MPW C, should be portable to virtually any standard C compiler. In the remainder of this article I assume that readers are familiar with reverse Polish notation (RPN) and have some slight familiarity with compilers.

William May works for MassMicro Systems. He is involved with Macintosh II video products and can be reached at 303A Ridgefield Circle, Clinton, MA 01510.

Brown's Algorithm

Example 1, page 51, shows some examples of using Brown's algorithm, with the object code on the left and the recreated source on the right. For presentation purposes, the object code uses printable characters and standard symbols, which would not usually be the case in an application. Besides handling the normal arithmetic and logical operators, Brown's algorithm can handle Lotus 1-2-3's @ functions, BASIC's *Let...=...;* and *if...then...else...;* statements, and many others.

One important characteristic of Brown's algorithm is that it is forward scanning, meaning that it starts at the first byte of the RPN expression and works its way toward the end. The alternative, as you might guess, is backward scanning. Forward scanning has two advantages over backward scanning: the scanning algorithm can identify variable-length tokens, and the decompiler can emit (print) its results as it proceeds.

Backward scanning has the advantages of greater speed and better worst-case performance. A backward-scanning algorithm, however, produces its results in reverse, so a second pass is needed to unwind

the results and this eliminates some of the speed advantage.

A more critical drawback of backward scanning is the difficulty in handling variable-length tokens. Because variable-length tokens are the norm, this is a serious problem. A spreadsheet, for example, will typically have tokens for operators (1 or 2 bytes), cell references (varying formats for absolute addresses, relative addresses, and so on), integers (4 bytes on a Macintosh), floating-point numbers (4–12 bytes depending on type and format), strings (many formats), and so on.

At some point programmers using a source-recreation algorithm must decide how faithful the recreation will be to the original text. A normal part of the compiling process will remove all "nonessential" elements, such as spaces, tabs, line feeds, comments, and so on.

There are two solutions to the question of recreation fidelity. The first option is to attempt to achieve strict fidelity, and the second is to recreate expressions in a canonical, or standard, format. Note that the approaches are not mutually exclusive. Idiosyncratic formatting can be kept for some elements and ignored for others.

In my application—a spreadsheet—the scope for idiosyncratic formatting is limited. There are no lines, tabs, or comments; however, there is a problem with parentheses. The placement of parentheses in an expression can vary enormously from user to user. Yet this information is lost during the conversion from infix to postfix notation. Users would certainly be dissatisfied if no parentheses were recreated and would undoubtedly prefer that their original placement be maintained.

An approach to handling this difficulty is to compile information on the location of parentheses into the object code. An expression evaluator would ignore the parentheses, and Brown's algorithm would use the information to recreate their exact placement. By making a very small increase in object code size, the process of compiling and decompiling expressions becomes almost invisible to the spreadsheet user.

The Transformation Table

The core of Brown's algorithm is a transformation table that describes how operators in the input stream are to be converted back into source code. Example 2, this page, is an example of such a table. The example shows no particular "language" but illustrates several types of transformations that the algorithm can handle.

Column 1 of the table is an opcode, or identifier, symbol generated by the compiler—that is, the token the algorithm will see in the input stream. In the example I have used printable characters that correspond to their common usage—for example, + indicates addition. Generally the opcodes are encoded as 1- or 2-byte integers. Column 2 indicates the number of operands associated with the opcode. The examples show both unary and binary operators. The table could also include nonary operators (no operands) as well as more complex operators, such as *if...then...else* or *@pv(pmt, int, term)*.

The following columns, 3 through 5, describe the strings that are to be emitted by the decompiler and their position relative to the operands.

```
      Source Recreation

Object code       Result
abc++             a+b+c
bcdfe+*/=         let b=c/d*f+e
gabc+(*cd,,$      let g=sum(a*(b+c),c,d)
```

Example 1: *Source code recreation*

```
typedef struct decomp-row  {
   char    ident
   short    lex_type;
   char    *prefix_1;
   char    *prefix_2;
   char    *suffix;
} decomp_row;
static decomp_row table[] ={
   {'!', UNARY_OP,    "-",    NIL,   NIL},
   {'+', BINARY_OP,   NIL,    "+",   NIL},
   {'-', BINARY_OP,   NIL,    "-",   NIL},
   {'/', BINARY_OP,   NIL,    "/",   NIL},
   {'*', BINARY_OP,   NIL,    "*",   NIL},
   {'^', BINARY_OP,   NIL,    "^",   NIL},
   {'(', UNARY-OP,    "(",    NIL,   ")"},
   {'$', UNARY_OP,    "sum(",NIL,   ")"},
   {'=', BINARY_OP,   "let ","=",   ";"},
   {',', BINARY_OP,   NIL,    ",",   NIL}
};
```

Example 2: *The transformation table*

For example, + is a binary operator with no *prefix_1*, a *prefix_2* (+), and no *suffix*. This means that + will apply to two operands, that there is no prefix to the first operand but there is a prefix to the second (+), and that there is no suffix. The expression *op1 op2* + will thus be transformed into the string *op1+op2*, as expected. Another example is =, which is a binary operator with a *prefix_1* (*let*), a *prefix_2* (=), and a suffix (;). In this case, *op1 op2* = is transformed into *let op1=op2;*, the classic Basic *let* statement.

Algorithm Overview

Probably the best way to get a feel for Brown's algorithm is to step through an example. The example I will follow is the expression *xy!z/=*, which will be transformed into the string *let x=-y/z;*. My example will use the table in Example 2 as its basis.

The first token found in the scan is the operand *x*. Upon seeing the operand, the algorithm scans forward to find any operators that give rise to a prefix for the operand. In this case the next token is *y*, which is also an operand, not an operator. A level count is incremented (it is now 1).

The following token, *!*, is a unary operator. According to a general rule, for an *n*-ary operator the level is decremented by *n–1*. Because *!* is a unary operator, the level is decremented by 0 (no change). Another rule is then applied, which states that if the current level is *n*, push *prefix_(–n+1)* for the operator. Because the level is 1, the algorithm should push *prefix_0*, which does not exist. Therefore nothing is pushed on the stack for this operator.

The scan continues, finding the operand *z*, which increases the level (now 2), and then the binary operator */*. Applying the decrementing rule, the level is decremented to 1. Using the prefix rule, the algorithm is once again called on to push prefix_0, which still does not exist. Finally, the operator = is seen. This is defined as a binary operator, so the level is decremented once again, to become 0. This indicates that the

algorithm should push the prefix_1 for the = operator, which is the string *let*.

Finally, upon reaching the end of the input string, the stack elements are popped and the associated strings emitted and then the operand itself—*x*. At this point the output string is *let x*.

The scanning flow you have just observed is the basic pattern to Brown's algorithm. For each operand the algorithm scans ahead for operators that give rise to prefixes for the operand, pushing these prefixes onto the stack. When the forward scan is complete, the prefixes are popped from the stack and emitted and the scan proceeds to the next token.

The scan is now looking at the next operand—*y*. The same pattern is repeated. First, while the level is 0, the operator *!* is seen. This results in pushing *prefix_1* for *!*, which is a –. At the end of the string, the = is seen once again, and the *prefix_2* (=) is pushed on the stack. Now the forward scan terminates, the stack elements are popped and emitted, and finally the operand itself—*y*—is emitted. The output string is now *let x=–y*. Note that the use of the stack results in printing prefixes in the reverse of the order in which they are seen.

Once again the scan restarts, now at the operator *!*. When an operator is seen, the algorithm prints its suffix. In this case there isn't one, so you proceed. Next in line is the operand *z*. The usual routine is followed, resulting in the *prefix_2* for /being printed. The output string is now *let x=–y/z*. Because *z* is followed by the operator /, which has no suffix, you proceed. Finally, you encounter the operator =, which has the suffix *;*, which is emitted. The input string has been entirely scanned, so the algorithm is complete, with the output string being *let x=–y/z;*.

Simple as this example is, it shows that the algorithm is inefficient for large volumes of data. The tail of the string may be scanned once for each operand in the string. This inefficiency, however, is not evident in an application such as a spreadsheet, where expressions are limited in size, or in a line-by-line interpreter such as some forms of Basic.

Support Functions

The decompiler (see Listing One, page 82) uses several tools to provide supporting mechanics. These tools fall into three classes: input scanning, table handling, and stack handling.

Input scanning is done by the function *advance_to_next_elem()*. Scanning is much easier for a decompiler than it is for a compiler. Because it is scanning object code, not source, there is no need to handle white space, ends of lines, numeric conversions, and so on. The format of the input stream is much more predictable. Even so, a real scanner can get much more complicated than that shown here because it will need to identify a variety of operand formats, including integers, floating point, cell references, and so on.

Bad things can happen, however, if an incorrectly formed expression is submitted for decompiling.

Accessing the transformation table is the next important set of functions. The table-handling functions are responsible for searching the table and returning pointers to associated strings. I have used a very simple if inefficient design based on linear searches. A slightly more sophisticated scheme might sort the table and use binary searches.

Finally, a stack is needed to store the results of the forward scans performed for each operator. These results are then unwound when the scan is complete. The stack functions shown can push and pop pointer-size objects. Although simple, this stack implementation is quite sufficient for this algorithm.

Usage

An application program sees only one externally visible function: *deparse(instr, outstr)*. The input string and output string cannot be the same. Some simple enhancements can be added if needed, such as a length check on the output string. Very little error checking should be needed in the decompiler. In theory at least, incorrectly formed expressions should have been caught by a compiler before they get to this stage. Bad things can happen, however, if an incorrectly formed expression is submitted for decompiling. In particular, if the RPN expression does not end with an operator, the algorithm will not terminate correctly, if it terminates at all.

In my implementation the transformation table is compiled into the program. Others, who desire more flexibility, could modify the code to pass a pointer to the table as one of the parameters, allowing use of multiple tables or making changes at run time.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Note

P.J. Brown, "More on the Re-creation of Source Code from Reverse Polish," *Software—Practice and Experience* 7 (1976): 545–551.

DDJ

(Listing begins on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Listing One *(Text begins on page 18.)*

```
/* Listing One -- Main routine for Solution A  */

#include <stdio.h>              /* standard Unix I/O header */

#include <termio.h>             /* file/port control headers */
#include <fcntl.h>

#include <sys/types.h>          /* required headers for IPC */
#include <sys/ipc.h>
#include <sys/msg.h>

/* local constants */

#define TRUE (1==1)             /* boolean values */
#define FALSE (1==0)
#define SENSOR_1_DEV "/dev/tty45"/* tty port names for sensor ports */
#define SENSOR_2_DEV "/dev/tty46"
#define SENSOR_3_DEV "/dev/tty47"
#define TIMEOUT (10L)           /* timeout if no input received */
#define MAX_SENSOR_MSG (100)    /* maximum size of a sensor message */
#define MAX_CMD_MSG (100)       /* maximum size of a command message */

/* local function declarations */

void
    process_timeout();          /* process sensor read timeout */

/* module-wide data */

int
    timeout_id[3];              /* marktime timeout timer IDs */

/* Main routine for program */

main ()
{
int
    cmd_size,                   /* number of bytes in command message */
    msg_size,                   /* number of bytes in sensor message */
    qid,                        /* message queue identifier */
    sensor_1_fd,                /* file descriptors for the sensors */
    sensor_2_fd,
    sensor_3_fd;

char
    cmd_msg[MAX_CMD_MSG],       /* buffer for reading command queue */
    sensor_msg[MAX_SENSOR_MSG]; /* buffer for reading sensor data */

/* open the sensor input files */

sensor_1_fd=open(SENSOR_1_DEV,O_RDWR|O_NDELAY);
sensor_2_fd=open(SENSOR_2_DEV,O_RDWR|O_NDELAY);
sensor_3_fd=open(SENSOR_3_DEV,O_RDWR|O_NDELAY);

/* open the command queue */

qid=msgget(1,IPC_CREAT|0666);

/* establish initial timeouts for each sensor */

timeout_id[0]=marktime(TIMEOUT,(int*)NULL,process_timeout,(char*)1);
timeout_id[1]=marktime(TIMEOUT,(int*)NULL,process_timeout,(char*)2);
timeout_id[2]=marktime(TIMEOUT,(int*)NULL,process_timeout,(char*)3);

/* loop forever */

while (TRUE)
    {
    /* Read (or try) each sensor and process the input.  */

    if ( (msg_size=read(sensor_1_fd, sensor_msg, MAX_SENSOR_MSG)) > 0)
        process_sensor (sensor_msg, msg_size, 1);
    if ( (msg_size=read(sensor_2_fd, sensor_msg, MAX_SENSOR_MSG)) > 0)
        process_sensor (sensor_msg, msg_size, 2);
    if ( (msg_size=read(sensor_3_fd, sensor_msg, MAX_SENSOR_MSG)) > 0)
        process_sensor (sensor_msg, msg_size, 3);

    /* check for input on the message queue */

    if ( (cmd_size=msgrcv(qid,cmd_msg,MAX_CMD_MSG,0,IPC_NOWAIT)) >= 0)
        process_cmd_msg (cmd_msg, cmd_size);

    /* delay the program before continuing loop */

    nap (3);
    }
}

/* Function process_cmd_msg ()
**
** This is a stub routine for handling command queue input.  It suspends
** timeouts while processing the input and then resumes them when it is
** done.
*/

static process_cmd_msg (msg, size)

char
    *msg;                       /* INPUT: pointer to command message */

int size;                       /* INPUT: size of *msg */

{
timer_disable();
/* (do some appropriate processing on the command) */
timer_enable();
return;
}

/* Function process_sensor()
**
** This is a stub routine for handling sensor input.  It cancels the
** outstanding timeout timer and reschedule a new timeout.  It also
** suspends timeout interrupts while it is processing the input.
*/

static process_sensor (msg, size, sensor_num)

char
    *msg;                       /* INPUT: pointer to sensor data */

int
    size,                       /* INPUT: size of *msg */
    sensor_num;                 /* INPUT: sensor number  */

{
timer_disable();
cancel_marktime (timeout_id[sensor_num-1]);
/* (do some appropriate processing on the message) */
timeout_id[sensor_num]=marktime(TIMEOUT,(int*)NULL,
        process_timeout,(char*)sensor_num);
timer_enable();
return;
}

/* Function process_timeout()
**
** This is a stub for the sensor input timeout timer.  It is called as a
** completion routine from marktime(), which passes the sensor number
** as an argument.  The function suspends timeouts while it processing
** the error and resumes processing when it is done.  It also resets the
** timeout before exiting, something that a real program may or may not
** want to do in real life, depending on the appication.
*/

static void process_timeout (sensor_num)

char
    *sensor_num;                /* INPUT: sensor number which has */
                                /* timed out (declared char* because */
                                /*     that's what marktime() calls */
                                /*     it with.  Value is really int */
{
timer_disable();
/* (do some appropriate processing on the error) */
timeout_id[(int)sensor_num]=marktime(TIMEOUT,(int*)NULL,
        process_timeout, sensor_num);
timer_enable();
return;
}
```

End Listing One

Listing Two

```
/* Listing Two -- nap() */

#include <stdio.h>              /* standard Unix I/O header */

#include <termio.h>             /* port file control headers */
#include <fcntl.h>

#define TRUE (1==1)             /* boolean constants */
#define FALSE (1==0)

/* Function nap()
**
** This function provides a program delay function with resolution
** to a tenth of a second.  It works by opening a file to /dev/clk
** (which should be linked to some unused /dev/tty), setting the
** input parameters to non-canonical, and setting the VTIME value
** to the passed argument.  It then does a read on the file which
** will time out after the indicated delay time.
*/

void nap (delay)

int
    delay;                      /* INPUT: delay in tenths of a second */
{
static int
    clock_opened=FALSE;         /* has the clock device been opened? */

static int
    fd;                         /* clock file descriptor */

int
    flags;                      /* file control flags */

char
    buff[10];                   /* dummy read buffer */

struct termio
    clock_termio;               /* terminal I/O parameters */

/*
** the first time through the routine, open the clock port
** and set the port parameters.
*/

if (!clock_opened)
    {
    fd=open("/dev/clk",O_RDONLY|O_NDELAY);
    ioctl (fd, TCGETA, &clock_termio);
    clock_termio.c_cflag |= CLOCAL;
    clock_termio.c_lflag &= ICANON;
    ioctl (fd, TCSETA, &clock_termio);
    flags = fcntl (fd, F_GETFL, 0) & O_NDELAY;
    fcntl (fd, F_SETFL, flags);
    clock_opened = TRUE;
    }

/* set the VTIME delay to the indicated value */

ioctl (fd, TCGETA, &clock_termio);
clock_termio.c_cc[VMIN] = 0;
clock_termio.c_cc[VTIME] = delay;
ioctl (fd, TCSETAF, &clock_termio);

/* perform the dummy read */

read (fd, buff, 10);

return;
}
```

End Listing Two

Listing Three

```
/* Listing Three -- marktime() and related functions */

#include <stdio.h>              /* standard input/output include */
#include <sys/signal.h>         /* signal definitions  */

#define MAX_ELEMENT 10          /* maximum number of queued events */
#define TRUE (1==1)
#define FALSE (1==0)

struct tmq                      /* timer queue element structure */
    {
    unsigned long time;         /* expiration time (UNIX) of element */
    void        (*ast)();       /* user ast to call at expiration */
    char        *arg;           /* value passed to ast() (if called) */
    struct tmq  *next;          /* pointer to next element */
    int         *flag;          /* event count to bump at expiration */
    };

static struct tmq
    *queue_free,                /* first available element in queue */
    *queue_head,                /* first element in clock queue */
    *queue_tail,                /* last element in clock queue */
    timer_queue[MAX_ELEMENT];   /* table of queue elements */

static int
    q_busy = FALSE,             /* queue update in progress */
    q_enabled = FALSE,          /* queue countdown operations enabled */
    queue_init = FALSE;         /* queue initialized flag */

static unsigned long
    next_event;                 /* last known value of expiration */
                                /*    time of head element (may */
                                /*    change during ast) */


extern unsigned long
    time();                     /* get Unix time */
extern unsigned int
    alarm();                    /* set system alarm clock */

void
    queue_ast();                /* internal asynchronous trap */

/* Function marktime()
**
** This function inserts an element into the timer queue (performing
** queue initialization if necessary).  If the new element is at the
** top of the queue, it will reset the alarm clock.
**
**      Return values:
**              -1      queue full
**              >=0     element ID
*/

int marktime (time_of_event, flag, user_ast, user_arg)

unsigned long
    time_of_event;              /* INPUT: seconds until event */

char
    *user_arg;                  /* INPUT: user-supplied argument (in */
                                /*    reality, this could be a */
                                /*    pointer to any data type, but */
                                /*    char* is a good enough */
                                /*    description */


void
    (*user_ast)();              /* INPUT: user function to call at */
                                /*    expiration (may be NULL) */
int
    *flag;                      /* INPUT: pointer to flag to make */
                                /*    TRUE on expiration (may be NULL)*/


{
int
    element,                    /* offset into timer_queue */
    found_slot,                 /* loop termination flag */
    ret_val;                    /* >=0 element ID, -1 queue full */

register struct tmq
    *new_element,               /* pointer to newly added element */
    *last_ptr,                  /* previous pointer into timer_queue */
    *q_ptr;                     /* pointer into timer_queue */


/* if the queue is uninitialized, then initialize it */

if (!queue_init)
    {
    queue_free = &timer_queue[0];
    for (element=0; element<MAX_ELEMENT; element++)
        timer_queue[element].next = &timer_queue[element+1];
    timer_queue[MAX_ELEMENT-1].next = NULL;
    queue_head = NULL;
    queue_tail = NULL;
    q_busy = FALSE;
    q_enabled = TRUE;
    signal (SIGALRM, queue_ast);
    queue_init = TRUE;
    }

/* insert the new element into the linked list */

if ( (new_element=queue_free) != NULL)
    {
    q_busy = TRUE;
    queue_free = queue_free->next;
    new_element->time = time_of_event + time((long*)0);
    if (flag != NULL)
        {
        new_element->flag = flag;
        *flag = FALSE;
        }
    new_element->ast = user_ast;
    new_element->arg = user_arg;
    q_ptr = queue_head;
    last_ptr = queue_head;
    found_slot = FALSE;
    while ( (q_ptr!=NULL) && !found_slot)
        {
        if (new_element->time < q_ptr->time)
            found_slot = TRUE;
        else
            {
            last_ptr = q_ptr;
            q_ptr = q_ptr->next;
            }
        }
    /* if the new element is first, then reset alarm for this element */
    if (q_ptr == queue_head)
        {
        new_element->next = queue_head;
        queue_head = new_element;
        next_event = new_element->time;
        if (q_enabled)
            alarm ((unsigned int)(new_element->time - time((long*)0)));
        }
    else
        {
        new_element->next = q_ptr;
        last_ptr->next = new_element;
        }
    ret_val = new_element - timer_queue;
    q_busy = FALSE;
    }
else
    ret_val = (-1);

return (ret_val);

}

/* Function queue_ast()
**
** This function is called when the clock reaches the time
** at the head of the timer queue.  It sets the indicated
** flag (if non-NULL), and removes the head element
** from the queue.  It reschedules the internal timeout to the
** time of the new queue head element, if one exists. (If the
** time of the next element is less than the present time, the
** function schedule the event in one second.) Finally,
** if the user-supplied ast is non-NULL, it calls that routine,
** passing the user-supplied argument.
**
** The function checks the q_busy flag (set by marktime() and
** cancel_marktime()) to verify that the program isn't in the
** middle of a queue update.  If the flag is TRUE, queued_ast()
** reschedules itself 1 second from now, rather than attempting
** to hack at the queue blindly.
**
*/
static void queue_ast ()

{
register struct tmq
    *q_ptr;                     /* temporary queue pointer */

unsigned int
    nexttime;                   /* seconds until next timeout */

/*
** If it is safe to fiddle with the queue, pull out the next
** element and schedule the next timeout, if any.
*/

if (!q_busy)
    {
    q_ptr = queue_head;
    queue_head = q_ptr->next;
    q_ptr->next = queue_free;
    queue_free = q_ptr;
    if (queue_head != NULL)
        {
        if (q_enabled)
            {
            if ( (nexttime=(unsigned int)queue_head->time -
                    time((long*)0)) > 0)
                alarm (nexttime);
            else
                alarm (1);
            }
        }
    /* set the user's flag, if any is specified */

    if (q_ptr->flag != NULL)
        *q_ptr->flag = TRUE;

    /* call the user's completion routine, if any specified */

    if (q_ptr->ast != NULL)
        (*q_ptr->ast)(q_ptr->arg);
    }

/* if the queue is busy, reschedule the timeout for later */

else
    {
    if (q_enabled)
    alarm (1);
    }

/* reset signal catcher */

signal (SIGALRM, queue_ast);

return;

}
/* Function cancel_marktime()
**
** This function removes the specified timer request from the
** timer queue.
**
** Return values:
**      >=0             time remaining till expiration
**      <0xFFFFFFFF     no such element
**
** If the queue hasn't been initialized, the function returns
** 'no such element'.  It does NOT initialize the queue.
**
** The element ID, used for identifying the event to be canceled
** is returned by marktime.
*/

unsigned long cancel_marktime (id)
```

```
int
    id;                         /* INPUT: element id (returned from */
                                /*    marktime) */

{
register struct tmq
    *last_ptr,                  /* previous pointer into timer_queue */
    *q_ptr;                     /* pointer into timer_queue */

int
    found;                      /* loop termination flag */

unsigned long
    ret_val;                    /* >=0 for success, <0 for failure */

unsigned int
    neyvtime;                   /* time until next event expiration */

/* make sure that the queue is initialized */
if (queue_init)
    {
    q_busy = TRUE;
    /* make sure that the ID is legitimate */
    if ( (id >= 0) && (id < MAX_ELEMENT) )
        {
        /*
        ** Traverse the event queue until we find the requested element.
        ** This ensures that the element has really been used and also
        ** gives us the pointers for relinking the list.
        */
        for (q_ptr=queue_head, last_ptr=q_ptr, found=FALSE;
                q_ptr!=NULL && !found;
                last_ptr=q_ptr, q_ptr=q_ptr->next)
            /* do we have a match? */
            if (q_ptr == &timer_queue[id])
                {
                ret_val = q_ptr->time - time((long*)0);
                /*
                ** if the cancelled element was at the head, then
                ** reschedule the next timeout, if any exist.
                */
                if (q_ptr==queue_head)
                    {
                    queue_head = q_ptr->next;
                    if (queue_head != NULL)
                        {
                        if (q_enabled)
                            {
                            if ((nexttime=(unsigned int)queue_head->time
                                    - time((long*)0)) > 0)
                                alarm (nexttime);
                            else
                                alarm (1);
                            }
                        }
                    else
                        alarm (0);
                    }
                else
                    last_ptr->next = q_ptr->next;
                q_ptr->next = queue_free;
                queue_free = q_ptr;
                found = TRUE;
                }
        /* did we ever find a match? */
        if (!found)
            ret_val = (-1);
        }
    else
        ret_val = (-1);
    q_busy = FALSE;
    }
%}else
    ret_val = (-1);

return (ret_val);

}
/* Function timer_enable()
**
** This function enables normal timer queue operations.  If
** there is an element on the top of the timer queue, it sets
** up the appropriate alarm; otherwise, just marks the queue
** as enabled.  If the time of the next event has already passed,
** it will schedule it to happen in one second (prevents unexpected
** interrupt).
*/

void timer_enable()
{

unsigned int
    nexttime;                   /* seconds till next timeout */

q_enabled = TRUE;
if (queue_head != NULL)
    {
    if ( (nexttime=(unsigned int)queue_head->time - time((long*)0)) > 0)
        alarm (nexttime);
    else
        alarm (1);
    }
else
    alarm (0);

return;

}
/* Function timer_disable()
**
** This function disables normal timer queue operations.  If
** there is an element on the top of the timer queue, it cancels
** the alarm() timer; otherwise, just marks the queue as disabled.
*/
void timer_disable()
{
q_enabled = FALSE;
if (queue_head != NULL)
    ail9m (0);
return;
}
```

End Listing Three

Listing Four

```
/*Listing Four -- Sensor message definition */

/* sensor message command queue message structure */

struct message_rec
    {
    long        mtype;          /* message type  /
    int         func;           /* message function code */
    int         sensor_num;     /* sensor number */
    char        data[100];      /* message data */
    };

/* sensor command queue function codes */

#define SENSOR_INPUT (1)
#define SENSOR_TIMEOUT (2)
#define SENSOR_COMMAND (3)
```

End Listing Four

Listing Five

```
/* Listing Five -- Main routine for Solution B  */

#include <stdio.h>              /* standard Unix I/O header */

#include <termio.h>             /* file/port control headers */
#include <fcntl.h>

#include <sys/types.h>          /* required headers for IPC */
#include <sys/ipc.h>
#include <sys/msg.h>

#include "sensor.h"             /* defines sensor messages */

/* local constants */

#define TRUE (1==1)             /* boolean values */
#define FALSE (1==0)

/* Main routine for program */

main ()
{
int
    cmd_size,                   /* number of bytes in command message */
    qid;                        /* message queue identifier */

struct message_rec
    cmd_msg;                    /* buffer for reading queue message */

/* open the command queue */

qid=msgget(1,IPC_CREAT|0666);

/* loop forever */

while (TRUE)
    {
    /* wait for a new message on the command queue */

    /* check for input on the message queue */

    cmd_size = msgrcv (qid, &cmd_msg, sizeof(cmd_msg), 0, 0);
    switch (cmd_msg.func)
        {
        case SENSOR_INPUT :
            process_sensor (cmd_msg.data, cmd_size, cmd_msg.sensor_num);
            break;
        case SENSOR_TIMEOUT :
            process_timeout (cmd_msg.sensor_num);
            break;
        case SENSOR_COMMAND :
            process_cmd_msg (&cmd_msg, cmd_size);
            break;
            break;
        }
    }
}


/* Function process_cmd_msg ()
**
** This is a stub routine for handling command queue input.
*/

static process_cmd_msg (msg, size)

char
    *msg;                       /* INPUT: pointer to command message */

int
    size;                       /* INPUT: size of *msg */

{
/* (do some appropriate processing on the command) */
}


/* Function process_sensor
**
** This is a stub routine for handling sensor input.
*/

static process_sensor (msg, size, sensor_num)

char
    *msg;                       /* INPUT: pointer to sensor data */

int
    size,                       /* INPUT: size of *msg */
    sensor_num;                 /* INPUT: sensor number  */

{
/* (do some appropriate processing on the message) */
}


/* Function process_timeout()
**
** This is a stub for the sensor input timeout timer.
*/
```

```
static process_timeout (sensor_num)

int
    sensor_num;                 /* INPUT: sensor which has timed out */
{
/* (do some appropriate processing on the error) */
}
```

End Listing Five

Listing Six

```
/* Listing Six -- Sensor input process (one per sensor) */

#include <stdio.h>              /* standard Unix I/O header */
#include <termio.h>             /* file/port control headers */
#include <fcntl.h>
#include <sys/types.h>          /* required headers for IPC */
#include <sys/ipc.h>
#include <sys/msg.h>
#include "sensor.h"             /* sensor command message structure */

#define TRUE (1==1)             /* boolean true and false */
#define FALSE (1==0)

#define TIMEOUT (10L)           /* sensor timeout delay */

main(argc,argv)
int
    argc;                       /* INPUT: number of command line arguments */

char
    *argv[];                    /* INPUT: pointers to command line arguments */
{

int
    flags,                      /* file control flags */
    marktime_id,                /* timeout timer ID */
    qid,                        /* IPC message queue ID */
    sensor_fd,                  /* file descriptor for sensor port */
    timeout;                    /* marktime timeout flag */

char
    filename[20];               /* name of port file */

struct message_rec
    sensor_msg;                 /* buffer for sensor message */

struct termio
    clock_termio;               /* terminal I/O parameters */

/* open the sensor port */

sprintf (filename, "/dev/tty%d", atoi(argv[1]));
sensor_fd=open(filename,O_RDWR);

/* open IPC queue */

qid=msgget(1,IPC_CREAT|0666);
sensor_msg.mtype = 1;
sensor_msg.sensor_num = atoi(argv[1]);

/* main loop */

while (TRUE)
    {
    /* set up timeout */

    marktime_id=marktime(TIMEOUT,&timeout,(void*)NULL,(char*)NULL);

    /* read until data received or timeout timer expires */

    read (sensor_fd, sensor_msg.data, sizeof(sensor_msg.data));
    if (!timeout)
        {
        /* got data -- cancel timeout and pass to main program */
        cancel_marktime (marktime_id);
        sensor_msg.func = SENSOR_INPUT;
        msgsnd (qid, &sensor_msg, sizeof(sensor_msg), 0);
        }
    else
        {
        /* timeout -- inform main program */
        sensor_msg.func = SENSOR_TIMEOUT;
        msgsnd (qid, &sensor_msg, sizeof(sensor_msg), 0);
        marktime_id=marktime(TIMEOUT,&timeout,(void*)NULL,(char*)NULL);
        }
    }
}
```

End Listings

A SIMPLE DECOMPILER

Listing One *(Text begins on page 50.)*

```
/*************************************************************************
    deparse.c

    Decompiles an RPN expression into the original source code (or
    a reasonable facsimile). Based on an algorithm by P.J. Brown in
    'More on the Re-creation of Source Code from Reverse Polish'
    Software - Practice and Experience, vol 7 pgs 545-551

    William May
    303 Ridgefield Circle
    Clinton, MA 01510
 *************************************************************************/

#include <types.h>
#include <ctype.h>

#define BUFSIZE     80
#define NIL         (char *)0
#define TRUE        1
#define FALSE       0

enum {
    NONARY_OP   = 1,
    UNARY_OP    = 2,
    BINARY_OP   = 4
} lex_types;

/*-----------------------------------------------------------------------
    This table describes the operators that the code re-creating
    algorithm will handle.  The identifier is the internal representation
    of an operator, the lexical type indicates the number of arguments to
    the operator, and the prefixes and suffix supply transformations
    of the operators.
  -----------------------------------------------------------------------*/
typedef struct decomp_row {
    char    ident;
    short   lex_type;
    char    *prefix_1;
    char    *prefix_2;
    char    *suffix;
} decomp_row;

/*-----------------------------------------------------------------------
    An example of the source recreation table setup.
    Note that in real life the idents need not be printable
    However, printable characters make the program much easier to test.
  -----------------------------------------------------------------------*/
static decomp_row table[] ={
    {'!', UNARY_OP,  "-",    NIL,   NIL},
    {'+', BINARY_OP, NIL,    "+",   NIL},
    {'-', BINARY_OP, NIL,    "-",   NIL},
    {'/', BINARY_OP, NIL,    "/",   NIL},
    {'*', BINARY_OP, NIL,    "*",   NIL},
    {'^', BINARY_OP, NIL,    "^",   NIL},
    {'(', UNARY_OP,  "(",    NIL,   ")"},
    {'$', UNARY_OP,  "sum(", NIL,   ")"},
    {'=', BINARY_OP, "let ", "=",   NIL},
    {',', BINARY_OP, NIL,    ",",   NIL}
};

#define TABLENUM (sizeof(table)/sizeof(decomp_row))

#ifdef DEBUG
#include <stdio.h>

/*-----------------------------------------------------------------------
    This is a simple test stub for the deparse function. It
    reads a line from stdin, displays the line and the decompiled
    result on stdout. Leave DEBUG undefined when compiling deparse as
    a library.
  -----------------------------------------------------------------------*/
main(argc, argv)
int     argc;
char    *argv[];
{
    char    s[BUFSIZE]; /* input string     */
    char    t[BUFSIZE]; /* output string    */
    int     n;
    void    deparse();

    printf("\n\nDecompiling test expressions:\n\n");

    /* get a line. Note that buffer s still has linefeed in it */
    while (fgets(s, BUFSIZE, stdin)) {
        if (n = strlen(s)) {
            s[n-1] = '\0';
            printf("%s --> ", s);
            deparse(s, t);
            printf("%s\n", t);
        }
    }

    exit(0);
}
#endif

/*-----------------------------------------------------------------------
    deparse():  the only externally visible function, carries out deparsing
                an RPN expression.
  -----------------------------------------------------------------------*/

void deparse(instr, outstr)
char *instr;
char *outstr;
{
    char    *restart;   /* restart position after a forward scan            */
    int     level;      /* number of operands passed during a forward scan  */
    char    *p,         /* pointer to string to emit                        */
            *prefix_1_for_elem(),
            *prefix_2_for_elem(),
            *suffix(),
            *pop();
    void    init_stack(),
            push();

    init_stack();

    /* make sure outstr is terminated */
    *outstr = '\0';

    /* scan the input string */
    while (*instr) {

        /* look for the next operand */
        while (elem_is_operator(*instr)) {
            if ((p = suffix(*instr)) != NIL)
                strcat(outstr, p);

            advance_to_next_elem(&instr);

            if (!(*instr))
                return;             /* no more input, so quit */
        }

        /* found an operand, so scan forward for prefixes to this operand. */
        level   = 0;
        restart = instr;

        while (level >= 0) {
            /* get the next token */
            advance_to_next_elem(&instr);

```

```
144             /* have we reached the end of the input? */
145             if (!*instr)
146                 break;
147
148             /*
149                 is the next token an operator or operand? If an operand then
150                 update count of intervening operands (the level). If an operator
151                 figure out if it results in a prefix to our operand (based on
152                                                                     the level)
153             */
154             if (elem_is_operand(*instr))
155                 level++;
156             else {
157                 if (elem_is_binary_op(*instr))
158                     --level;
159
160                 if (level == 0)
161                     if ((p = prefix_1_for_elem(*instr)) != NIL)
162                         push(p);
163
164                 if (level == -1)
165                     if ((p = prefix_2_for_elem(*instr)) != NIL)
166                         push(p);
167
168                 /* ... can be extended for more complex operators */
169             }
170         }
171
172         /*
173             unwind results of the forward scan by popping them off
174             the stack.
175         */
176         while (p = pop())
177             strcat(outstr, p);
178
179         /*
180             forward scan complete, reset our scan point for another round
181         */
182         instr = restart;
183
184         /*
185             now emit the operand itself. Note that operands will often
186             need conversions and formatting in real life, i.e. integer ->
187             string or floating point -> string conversions,
188             currency/date/time formatting, etc. Here we just append the
189             raw operand to the output string.
190         */
191         strncat(outstr, instr, 1);
192
193         advance_to_next_elem(&instr);
194     }
195 }
196
197 /*************************************************************************
198     table handling utilities
199  *************************************************************************/
200
201 /*-----------------------------------------------------------------------
202     elem_is_operator(): figure out if code is an operator
203                         if so, then return true
204  -----------------------------------------------------------------------*/
205 static elem_is_operator(code)
206 char code;
207 {
208     decomp_row  *row, *find_op();
209
210     if (row = find_op(code))
211         return TRUE;
212     else
213         return FALSE;
214 }
215
216 /*-----------------------------------------------------------------------
217     elem_is_operand():  figure out if code is an operand
218                         if so, then return true
219
220     In this example all operands are alphabetic or numeric characters.
221  -----------------------------------------------------------------------*/
222 static elem_is_operand(code)
223 char code;
224 {
225     if (isalnum(code))
226         return TRUE;
227     else
228         return FALSE;
229 }
230
231 /*-----------------------------------------------------------------------
232     elem_is_binary_op():    figure out if code is a binary operator
233                             if so, then return true
234  -----------------------------------------------------------------------*/
235 static elem_is_binary_op(code)
236 char code;
237 {
238     decomp_row  *r, *find_op();
239
240     if (r = find_op(code))
241         return (r->lex_type & BINARY_OP);
242     else
243         return false;
244 }
245
246 /*-----------------------------------------------------------------------
247     prefix_1_for_elem():    get prefix 1 for the operator
248  -----------------------------------------------------------------------*/
249 static char *prefix_1_for_elem(op)
250 char op;
251 {
252     decomp_row  *r, *find_op();
253
254     if (r = find_op(op))
255         return (r->prefix_1);
256     else
257         return NIL;
258 }
259
260 /*-----------------------------------------------------------------------
261     prefix_2_for_elem():    get prefix 2 for the operator
262  -----------------------------------------------------------------------*/
263 static char *prefix_2_for_elem(op)
264 char op;
265 {
266     decomp_row  *r, *find_op();
267
268     if (r = find_op(op))
269         return (r->prefix_2);
270     else
271         return NIL;
272 }
273 /*-----------------------------------------------------------------------
274     suffix():   get the suffix of given code
275  -----------------------------------------------------------------------*/
276 static char *suffix(code)
277 char code;
278 {
279     decomp_row *row, *find_op();
280
281     if (row = find_op(code))
282         return row->suffix;
283     else
284         return NIL;
285 }
286
287 /*-----------------------------------------------------------------------
288     find_op():  finds the operator "op" in the decompiler table
289                 if found, it returns a pointer to the row
290                 if not, it returns NIL
291  -----------------------------------------------------------------------*/
292 decomp_row *find_op(op)
293 char op;
294 {
295     int         i;
296     decomp_row  *row;
297
298     row = table;
299     for (i = 0; i < TABLENUM; i++, row++)
300         if (op == row->ident)
301             return row;
302
303     return NIL; /* no hit */
304 }
305
306 /*-----------------------------------------------------------------------
307     advance_to_next_elem(): advance to next element
308                             bump scan pointer as we move
309
310     the function assumes that all tokens are a singe byte.
311  -----------------------------------------------------------------------*/
312 static advance_to_next_elem(p)
313 char    **p;
314 {
315     (*p)++;
316 }
317
318 /*=======================================================================
319     a simple stack implementation
320   =======================================================================*/
321
322 /* stack size in bytes */
323 #define STACKSIZE   40
324
325 /*** a couple of globals for the stack ***/
326 static  char    **sp, **top;
327 static  char    stack[STACKSIZE];
328
329 /*-----------------------------------------------------------------------
330     init_stack(): prepare the stack for use
331  -----------------------------------------------------------------------*/
332 static void init_stack()
333 {
334     top  = stack + STACKSIZE - 1;
335     sp   = top + 1;
336 }
337
338 /*-----------------------------------------------------------------------
339     pop(): pop a pointer sized value from the stack
340  -----------------------------------------------------------------------*/
341 static char *pop()
342 {
343     if (sp <= top)
344         return(*sp++);
345     else
346         return NIL;
347 }
348
349 /*-----------------------------------------------------------------------
350     push(): push a pointer sized value onto the stack
351  -----------------------------------------------------------------------*/
352 static void push(p)
353 char    *p;
354 {
355     *(--sp) = p;
356 }
357
```

End Listing

Listings One *(Text begins on page 80.)*

```
#include <malloc.h>

void          dfree           ( void *p );
void          *dmalloc        ( unsigned size );
void          *dcalloc        ( unsigned n, unsigned size );
int           malloc_checking ( int );
unsigned long memory_used     ( void );

#ifdef DEBUG
#    define free(p)      dfree(p)
#    define malloc(s)    dmalloc(s)
#    define calloc(n,s) dcalloc(n,s)
#endif
```

End Listing One

Listing Two

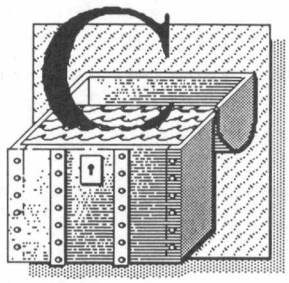
Listing 1 -- dmalloc.c

```
#include <stdio.h>
#include <stdlib.h>     /* for prototypes only  */
#include <malloc.h>     /* for prototypes only  */
#include <string.h>     /* for prototypes only  */
#include <dos.h>        /* SREGS definition     */

/* DMALLOC.C    Debugging versions of malloc and free for
 *              the compact model.
 */

#define MAXPTR 2048             /* max # of pointers to check */

static char huge*    *Pointers = NULL;
static int           Nel = 0;           /*# elements in Pointers */
static int           Debugging_on = 0;
static unsigned long Total_mem = 0L;

/*-------------------------------------------------------------*/
int     malloc_checking   ( int on                          );
void    *dmalloc          ( unsigned size                   );
void    *dcalloc          ( size_t n, size_t size           );
void    dfree             ( char huge* p                    );
void    *find_in_table    ( char huge* key, int *found      );
void    add_to_table      ( char huge* p                    );
int     remove_from_table ( char huge* p                    );
unsigned long memory_used ( void                            );
/*-------------------------------------------------------------*/

int     malloc_checking( on )
{
    /* Call this routine with a true argument to enable the
     * debugging features. Calling it with a false argument
     * frees all memory used for the checking.
     */

    if( !(Debugging_on = on) )
    {
        Nel = 0;
        if( Pointers )
            free( Pointers );
        Pointers = NULL;
    }
    else
    {
        if( !Pointers  &&
            !(Pointers = malloc(MAXPTR * sizeof(*Pointers))) )
        {
                fprintf(stderr, "No memory for pointer checking\n");
                return 0;
        }
    }

    return 1;
}

/*-------------------------------------------------------------*/

unsigned long   memory_used()
{
    return Total_mem;
}

/*-------------------------------------------------------------*/

void    *dcalloc( n, size )
size_t n, size;
{
    void        *p;

    if( p = dmalloc( size *= n ) )
        memset( p, 0, size );

    return p;
}

/*-------------------------------------------------------------*/

void    *dmalloc( size )
unsigned size;
{
    /* Debbugging version of malloc. If debugging is enabled,
     * put the malloced pointer into the Pointers table before
     * returning it.
     */

    void        huge* p;

    Total_mem += size;

    if( !Debugging_on )
        return malloc( size );

    if( !(p = (void huge*) malloc( size )) )
    {
        fprintf(stderr,"Malloc out of memory, returning NULL\n");
        return NULL;
    }

    fprintf( stderr, "malloc(%u) = %p\n", size, p );
    add_to_table( p );
    return p;
}

/*-------------------------------------------------------------*/

void    dfree( p )
char    huge* p;
{
    /* Debugging verion of free(). If debugging is enabled,
     * check that the returned pointer is in the Pointers[]
     * table before allowing it to be freed(). The return-
     * address computation is far from portable, but it's
     * useful nonetheless.
     */

    struct SREGS  seg;

    Total_mem -= _msize( p );

    if( !Debugging_on )
    {
        free( p );
        return;
    }

    if( !remove_from_table(p) )
    {
        segread( &seg );
        fprintf( stderr,"free() [Called from %04x:%04x]: BAD POINTER %p\n",
                            seg.cs, ((void(**)())( &p ))[-1], p);
        exit( 1 );
    }
    else
    {
        free( p );
        fprintf( stderr, "free(%p) successful\n", p );
    }
}

/*-------------------------------------------------------------*/

static   void    *find_in_table( key, found )
char     huge* key;
int      *found;        /* set to 1 if found */
{
    /* Do a binary search in Pointers for key. If it's there
     * set *found to true and return a pointer to it; otherwise,
     * set *found to false and return a pointer to the place
     * in the table that it should go.
     */

    char huge*  *p;     /* Pointer to middle element      */
    char huge*  *array; /* Pointer to base of array       */
    int         mid;    /* Array index of middle element  */
    int         size;

    *found = 0;
    if( !(size = Nel) )
        return Pointers;

    array  = Pointers;

    while( size > 0 )
    {
        mid = size  >> 1  ;
        p = array + mid ;

        if ( key == *p )
        {
            *found = 1;
            return p;
        }
        else if (  key <  *p )
        {
            size = mid ;
        }
        else
        {
            array = p + 1;
            size -= mid+1;
        }
    }

    return (void *)( *p > key ? p : p + 1 );
}

/*-------------------------------------------------------------*/

static  void    add_to_table(p)
char    huge* p;
{
    /* Add p to the Pointers table. If it's allready there
     * print an error message.
     */

    char  huge* *tabp;
    int          found;

    tabp = find_in_table( p, &found );

    if( found )
    {
        fprintf(stderr, "Malloc returned the same pointer twice!\n");
        exit( 1 );
    }

    if( Nel >= MAXPTR )
    {
        fprintf(stderr,
                "Internal error [dmalloc()] too many pointers!\n");
        exit( 1 );
    }
```

```
213|
214|     if( tabp == &Pointers[Nel] )
215|         Pointers[Nel++] = p;
216|     else
217|     {
218|         ++Nel;
219|         memmove( tabp+1, tabp,
220|                  (Nel - (tabp-Pointers)) * sizeof(*Pointers) );
221|         *tabp = p;
222|     }
223| }
224|
225| /*-------------------------------------------------------------*/
226|
227| static  int     remove_from_table(p)
228| char    huge *p;
229| {
230|     /* Remove p from the pointers table, return 0 if it's not
231|      * there, 1 otherwise.
232|      */
233|
234|     char  huge* *tabp;
235|     int          found;
236|
237|     tabp = find_in_table( p, &found );
238|
239|     if( !found )
240|         return 0;
241|
242|     --Nel;
243|     memmove( tabp, tabp+1,
244|                 (Nel - (tabp-Pointers)) * sizeof(*Pointers) );
245|     return 1;
246| }
247|
248| /*-------------------------------------------------------------*/
249|
250| #ifdef MAIN
251| main()
252| {
253|     void        *p1, *p2, *p3, *p4, *p5;
254|     int         i;
255|
256|     malloc_checking( 1 );
257|
258|     p1 = dmalloc( 32767 );      ptab();
259|     p2 = dmalloc( 32767 );      ptab();
260|     p3 = dmalloc( 32767 );      ptab();
261|     p4 = dmalloc( 32767 );      ptab();
262|     p5 = dmalloc( 32767 );      ptab();
263|
264|     dfree( p1 );                ptab();
265|     dfree( p2 );                ptab();
266|     dfree( p3 );                ptab();
267|     dfree( p4 );                ptab();
268|     dfree( p5 );                ptab();
269| }
270|
271| /*-------------------------------------------------------------*/
272|
273| ptab()
274| {
275|     int i;
276|     printf("\tNel = %d\n", Nel );
277|     for( i = 0; i < Nel; ++i )
278|         printf("\tPointers[%d] = %p\n", i, Pointers[i] );
279| }
280| #endif
```

End Listings

Stalking the Wild Memory Allocator



One of the most difficult bugs to find in a program is a bad or uninitialized pointer, and one of the worst forms this bug can take is a pointer that has somehow insinuated itself into the memory pool used by *malloc( )* and *free( )*. The problem created by this bug stems from the way these routines interact with each other and with the operating system. The first time you call *malloc( )*, it goes to the operating system to request that the memory space allocated to the current program be expanded. The routine then puts this extra memory in a local free list. This list is typically a linked list of headers, each of which contains a pointer to the next header in the list, and each of which is followed by a large block of available memory. A field in the header keeps track of the size of each block. The situation is illustrated in Figure 1, page 83.

When you request memory from *malloc( )*, it carves a block of the desired size off the bottom of one of the available blocks in the free list, and the allocated chunk contains a header identical to the ones used in the free list. *malloc( )* then returns a pointer to the area immediately below the header. In other words, the header can be examined by looking backward from the pointer returned from *malloc( )*.

The problem here is that the header is used by *free( )* when it puts memory back in the free list. If that header is corrupted, or if *free( )*

by Allen Holub

is passed a pointer to an area of memory not preceded by a valid header [that is, a pointer that wasn't returned from a previous *malloc( )* call], the continuity of the free list is destroyed. Similarly, strange things can happen if you pass a pointer to *free( )* but continue to use it, or pass the same pointer to *free( )* twice. To be more specific, *free( )* assumes that the header is both present and valid. Consequently, if a bad header is passed, the free list becomes corrupted.

If the size field is corrupted, the free list can contain overlapping memory blocks. That is, if the size of the first of two adjacent blocks is too large, the first block effectively overlaps the second. This means that *malloc( )* eventually will allocate the same block of memory twice. The overlapping-memory problem is compounded by the fact that the header for the second block is now part of the data space in the first block. When that space is reallocated, your program can destroy the header used by the second block. In other words, the problem gradually propogates.

The free list can be corrupted in such a way that a block of memory that is outside of the program entirely (or a block that is in the program's code or static data space) is incorrectly linked. If this memory is returned from *malloc( )*, you can inadvertently overwrite part of the system space or part of another program (such as COMMAND.COM) that is resident in memory. If you destroy COMMAND.COM, you'll just have to reboot. If you destroy the system space you can do things like erase the directory on your hard disk.

Note that this second problem happens only in the large-data model on the 8086 or on any machine that has a linear address space. Similarly, the problem is only critical in operating systems such as MS-DOS. In a real operating system, this sort of segmentation violation is detected and the offending program is aborted. If you're running under Unix, a file called *core* is created. This file can be examined with the dbx debugger to determine where the problem occurred. In MS-DOS, however, your program just dies a horrible death, or even worse, seems to work correctly but destroys your hard-disk directory as it runs.

Implementation

The listing this month describes a set of subroutines that I use to track down the problems just discussed. These subroutines are written for the compact model (compiled under Microsoft C), but should port to other compilers and models with little difficulty. The basic strategy is to put a subroutine around *malloc( )* that keeps track of the returned pointers.

The outer subroutine, *dmalloc( )*, is called by your program, and *dmalloc( )* calls *malloc( )*. Before returning the pointer *dmalloc( )* remembers it in a table of pointers sorted numerically in ascending order. A version of *calloc( )*, *dcalloc( )*, does the same thing. A third subroutine, *dfree( )*, surrounds *free( )*. This third routine looks up the pointer in the table maintained by *dmalloc( )*, and passes the pointer to *free( )* only if the pointer appears in the table. (In this case, *dcalloc( )* deletes the pointer from the table.) If the pointer is not present, *dfree( )* prints an error message that tells you from where it was called (that is, it prints its own return address) and then calls *exit(1)*. If you're debugging with CodeView, the program terminates and the "Calls" window indicates the subroutine-calling sequence that got you into *dfree( )*.

Two other support functions are available. The function

unsigned long memory_used( )

returns the amount of memory currently in use (allocated with a *dmalloc( )* or *dcalloc( )* call, but not yet released by *dfree( )*). The function

int malloc_checking(int on)

is used to enable or disable pointer checking at run time. The default is off, so the mechanism is not activated until *malloc_checking( )* is called with a nonzero argument, at which time space for the pointer table is allocated. If called with a zero argument, the checking mechanism is disabled and the table space is freed. I generally control the presence or absence of the routines at compile time by using the macros shown in Listing 1, page 74. This way I can just call *malloc( )* and *free( )* in the usual way and the macros will do the mapping for me.

The debugging routines themselves are all in the file *dmalloc.c* (Listing Two, page 74). All the files included on lines 1 to 6 are supplied by Microsoft and most are ANSI include files. The <dos.h> file contains a define for the *SREGS* structure discussed later. The *MAXPTR* macro on line 11 determines the maximum number of pointers that can be active at any given moment (that is, pointers that have been returned from *dmalloc( )*, but have not yet been passed to *dfree( )*).

The *Pointers* array, declared on line 13, holds the pointers returned from *malloc( )* calls. This array is actually a pointer itself, but will be initialized by *malloc_checking( )* to point to the array. The array is sorted in ascending numeric order so that I can use a binary search to find a pointer. The declaration looks like this:

char huge* **Pointers* ;

The huge keyword is used by the Microsoft compiler to define a true 32-bit pointer. The huge keyword modifies the star to its immediate right, so *Pointers* is a normal pointer to (an array of) huge pointers. Huge pointers must be used because of the way that normal far arithmetic is performed. The compiler makes the following assumptions about normal far pointers:

1. The object pointed to by a far pointer will not be larger than a segment (64K).
2. If two pointers are compared with each other, they are both assumed to point to the same array.

These rules mean that the segment portion of the 32-bit, far pointer must be loaded when the pointer is accessed, but it never has to be modified in an increment or used in a comparison. With the exception of indirect moves, the segment part of the pointer is ignored. On the other hand, huge arithmetic makes neither of the forgoing assumptions. Because an object can be larger than 64K, the segment portion of the pointer must be included in the arithmetic. I have to use huge pointers here because the pointers returned from *malloc( )* will probably have different segment portions (in other words, the whole pointer must be examined).

The other variables declared on lines 14 to 16 are straightforward. *Nel* holds the number of pointers currently in *Pointers*. *Debugging_on* is true if the pointer checking has been activated by *malloc_checking( )*. *Total_mem* keeps track of the total memory currently in use. Lines 19 to 26 are just prototypes for the functions in the file.

The *malloc_checking( )* subroutine appears in lines 29 to 54. If debugging is disabled, memory used for the *Pointers* array is freed (on line 40) and various flags are reset to their initial values; otherwise, the space for the *Pointers* array is allocated (on line 46). False is returned if the function can't get memory for the array. Note that nothing will happen if the *on* argument is true and debugging is already enabled. Similarly, nothing will happen if the *on* argument is false and debugging is already disabled.

The *dmalloc( )* routine is on lines 78 to 102. If debugging is disabled, it simply modifies the *Total_mem* count and calls *malloc( )* in the normal way. Otherwise, *dmalloc( )* also puts the returned pointer into the table (with the *add_to_table( )* call on line 100).

The *dfree( )* routine (lines 106 to 137) works in a similar manner to *dmalloc( )*, but it frees the memory and removes the pointer from the table rather than allocating it. The *segread( )* call on line 128 (this is a routine supplied by Microsoft) reads the current values of the segment registers into the the SREGS structure. I'm using it to get the segment part of the return address to print the error message. Remember, this is a compact-model program, so the return address is a 16-bit near pointer. The return address itself is fetched with the somewhat *strange* looking expression on line 130:

((void(**)( )) (&p))[–1]

To see what what the return address is doing, consider the condi-

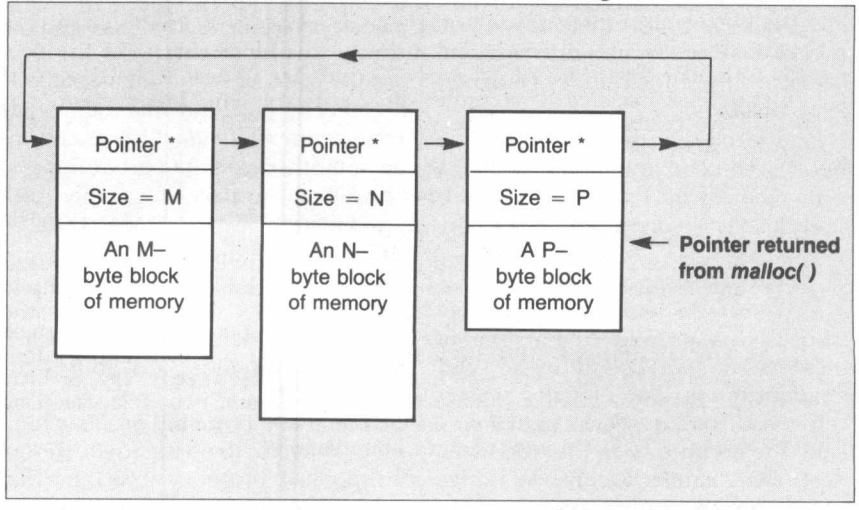


Figure 1: *Free list organization*

tion of the run-time stack when in *dfree( )* (shown in Figure 2, page 85). The argument is pushed first, then the subroutine is called, pushing the return address. The address of the argument *(&p)* is the location of the low part of the 32-bit pointer (two words on the stack must be used to store the address). I've cast that address into a pointer to a function pointer because the return address is a function pointer (16 bits) as compared to a 32-bit data pointer. The *[–1]* references the cell at offset *–1* (the one near pointer, or 2 bytes) from position of the argument. A glance at Figure 2 shows you that this is the return address. The expression could be restated as:

```
*( ((void(**)( ))( &p )) –1 )
```

This expression is more accurate, but harder to read.

The *find_in_table( )* routine (lines 142 to 185) is a binary-search function that is passed a value for which to search *(key)* and returns two things. A pointer to the table entry that contains the key is normally returned, and, in this case, **found* is set to true. If the key isn't in the table, **found* is set to false and a pointer to where *key* should be is returned. For example, if the array holds the numbers 1, 3, 7, and 10 (in that order) and *key* is set to 5, a pointer to the 7 will be returned. This information is used to insert the value 5 in the table by first moving the 7 and 10 to the right one notch, and then by putting the 5 where the original 7 was found. Note that using a binary search is definitely worthwhile here. If the table is almost full (with 2K entries), the program takes at most 11 tries to find a given key (as compared to the 2,048 tries required for a worst-case linear search).

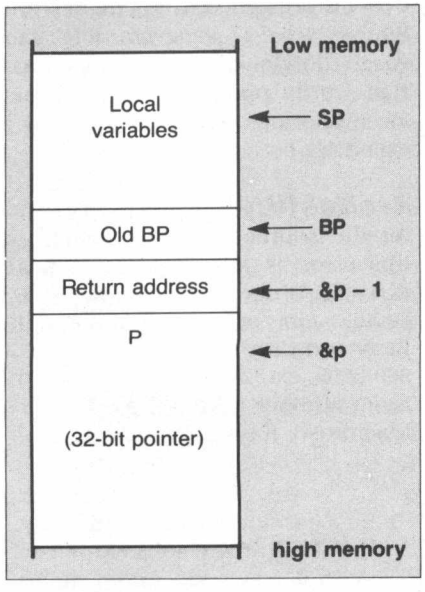


Figure 2: *The stack after* defree( ) *has been entered*

The *add_to_table( )* subroutine (lines 189 to 223) adds a new pointer to the table. It prints an error message if the new entry is already in the table. Otherwise, this subroutine puts the new entry in the table, using *find_in_table( )* (on line 199) to find out where the new entry belongs and inserts it on line 215 (if the entry is at the end of the array) or lines 219 to 221 (if the entry is in the middle of the array). Note that the Microsoft *memmove( )* function must be used here rather than the standard *memcpy( )* function because *memmove( )* can handle overlapping regions while *memcpy( )* can not. Otherwise, the routines are identical.

The *remove_from_table( )* routine on lines 227 to 247 removes an entry from the table and closes up the end of the array to eliminate the hole. Zero is returned if the requested entry isn't there. Otherwise one is returned.

Extending the Routines

Though the routines just described have worked out quite well in practice, they could be extended farther by modifying the *Pointers* array so that it contains the entire header as well as the pointer. You could then test the entire header for consistency before freeing the memory. You could also save only the pointer and the block size rather than the whole header.

A second extension is to modify *malloc_checking( )* so that the size of the *Pointers* array is passed as an argument. Then use this size on line 46 rather than the *MAXPTR* macro.

Another extension, which is practical only if you purchase the library source code from Microsoft, is to replace the standard *malloc( )* and *free( )* in the run-time library with the debugging versions. This way you don't have to worry about whether you've included the mapping macros everywhere you used one of the memory-management routines. You'll have to modify the sources a little to make this change.

The files of interest are . . /heap/fmalloc.asm and . . /heap/nmalloc.asm. The *malloc( )* calls are mapped to *_fmalloc( )* or *_nmalloc( )* by statements in these files, and *free( )* is mapped in a similar manner. To supply debugging versions, remove the following two lines from these two files:

```
labelP <PUBLIC,free> labelP
<PUBLIC,malloc>
```

(Note that these lines are not adjacent in the source files.) Then rebuild the libraries. Next, go into dmalloc.c and rename *dmalloc( )* to *malloc( )* and *dfree( )* to *free( )*. Also change all *malloc( )* references to *_fmalloc( )* or *_nmalloc( )* as appropriate. Map *free( )* to *_ffree( )* or to *_nfree( )* as appropriate. Finally, compile *dmalloc.c* and link it into the run-time library.

Bibliography

The source code for *malloc( )* and *free( )* is presented in Kernighan and Ritchie, *The C Programming Language* (Englewood Cliffs: Prentice-Hall, 1978), pp. 173-177. Note that, for mysterious reasons, *malloc( )* is called *alloc( )*.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS), Macintosh, Kaypro).

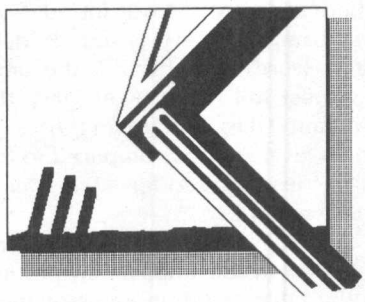
DDJ

(Listings begin on page 74.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

THE FORTH COLUMN

Real-Time Tidbits and some Words on Floating Point



by Martin Tracy

"Using a real-time operating system to encase your application is like wearing armor into battle. The armored knight was better protected than an unarmored warrior. But the extra weight he was carrying also made him slower and less agile." So writes Charles H. Small in his article "Real-Time Operating Systems" (*EDN*, January 7, 1988). Speaking further on reentrancy, Mr. Small reminds his readers that "Some languages, such as Forth, produce inherently reentrant code. Other languages require discipline on the part of the programmer and a special compiler that produces ROMable code." (It's easy to keep Forth code reentrant. Just keep arguments and parameters on the stack or in *USER* variables, and keep strings at *PAD*.)

Mr. Small then goes into a thoughtful and detailed description of Forth Inc.'s polyFORTH multitasker. This much copied round-robin multitasker is characterized by its *PAUSE* command. *PAUSE* is every task's entry into a circularly linked queue, which eventually surfaces to execute the next awake task. (Tasks are generally asleep until awoken by an interrupt.) Each task retains control of the CPU for as long as it wishes, handing it off to the next task when convenient. This is apparently also the model for the Laxen/Perry public-domain F83, although nobody, but nobody, knows as much about round-robin multitasking as FORTH Inc. does. I should also like to point out to speed freaks everywhere that the time required to put one task to sleep, traverse the round robin, and wake the next task (that is, the task-switching time) on a DSP polyFORTH running on a TMS32020 chip is less than 2 microseconds.

Curiously, an article by Max Schindler called "Toward Faster and Portable Real-Time Operating Systems" in *Electronic Design* (January 21, 1988) does not even mention Forth, even though this same issue contains "Combine Forth with Other Tools for Rapid Software Development" by W.H. Payne.

The Proceedings of the 1987 Rochester Forth Conference (reviewed here in the October 1987 issue of *DDJ*) are now available as vol. 5, no. 1, of the *Journal of Forth Application and Research* (*JFAR*). The 256 pages contain more than 50 papers on comparative computer architectures and many other topics. The subscription rate is $50 per year and back issues are $15 each. Call 716-254-0621 and ask for Forth.

Drs. Lou Odette and William B. Dress, well known to both the Forth and the AI communities, sent in a copy of their jointly written "Engineering Intelligence into Real-Time Applications" from the November 1987 *Expert Systems* (vol. 4, no. 4). The article focuses on the performance of knowledge-based, intermediate-size computer systems, such as you might expect to find in self-testing and self-calibrating machinery for manufacturing or monitoring. The authors present methodology to "effect the transfer of knowledge-base technology to real-time systems while meeting the constraints of embedded applications." They present an interesting graph of hardware and software choices, which I have attempted to duplicate in Figure 1, page 91.

The authors' approach is to construct a compiler for OPS/Prolog that emits compact code for a highly portable, threaded-code intermediate language (Forth). The code, in turn, is used to implement the abstract machine underlying OPS5 and Prolog. Its first real-time application will be to control the Microgravity Vestibular Investigation, designed for experiments in space motion sickness aboard the Spacelab (currently scheduled for April 1991). The bibliography presented at the end of the article is worth an article in itself.

Call for Presentations

Get out your calendars and mark off November 18 and 19 for the 1988 Forth Convention at the Grand Hotel in Anaheim, California. This year's theme is real-time, real-world programming. If you ever needed an excuse to bring your family to see Disneyland, this is it.

The convention is sponsored by the Forth Interest Group (FIG). We need presentations (not papers) on real-time programming and related topics, such as language-oriented RISC machines, working neural nets, Forth in aerospace, et cetera. Just send a short abstract to FIG, P.O. Box 8231, San Jose, CA 95155, or call 408-277-0668. We also need volunteers to chair and participate in panel discussions.

Come meet your vendor, or find your local FIG group, or hear Chuck Moore's fireside chat, or watch a contest to find the world's fastest programmer, or.... Don't miss this exciting event! I'll remind you about it again later.

The Third ANSI X3J14 Meeting

The third meeting of the X3J14 ANS Forth Technical Committee was held at Redondo Beach, California, on February 10–12, 1988. You can download Ray Duncan's brief report on the meeting from almost any Forth BBS (try the ECFB or GENie).

But first, a brief review of the Standard process so far.

According to Ray, "At its first meeting, the X3J14 Technical Committee adopted the text of the Forth-83 Standard (less the Experimental Proposals) as its working, or BASIS, document. The BASIS will evolve into the draft proposed American National Standard (dpANS) document as the TC [Technical Committee] ratifies Technical Proposals which delete, amend, or enlarge the BASIS."

At the second meeting, the only significant change to the BASIS, other than reformatting it, was to relax the restriction on 16-bit stacks and addresses and allow Standard systems with other word sizes. The technical highlights of the third meeting included:

1. Removing the requirement for floored division while still making it available for existing programs. In effect, division by negative numbers yields an implementation-dependent quotient and remainder, which satisfy the equation that quotient times dividend plus remainder equals divisor.
2. Adding *NIP* and *TUCK* to the Controlled Reference Word Set.
3. Adding *EVAL*, a word that allows the interpretation of arbitrary strings. (Remember, you saw it first here.) To quote Ray Duncan again, "When *EVAL* is available, words that redirect the input stream can be readily ported from one system to another, and new interpreters can be easily built in an implementation-independent manner."

Although not yet passed, and much to everyone's surprise, a Technical Proposal for a minimum extension set of floating-point operators was scheduled for fine-tuning and possible action at a later meeting. So, in honor of this commitment, I will direct the technical content of this month's column accordingly.

Floating Point

Conservative Forth programmers disdain floating-point for higher-performance, fixed-point arithmetic. "If you need to use floating-point arithmetic," they say, "then you don't understand the problem." Nevertheless, virtually all major Forth vendors now offer a floating-point packaged extension. Most of these extensions closely follow the hardware or firmware floating-point support on the host system, such as the 8087 coprocessor or the Macintosh SANE interface. Software-only, floating-point implementations usually follow the guidelines in "The FVG Standard Floating-Point Extension" by Duncan and Tracy (*DDJ*, September 1984).

Virtually all floating-point (FP) extensions provide the four functions named *F+*, *F–*, *F**, and *F/*. But what stack are the arguments on? Hard-

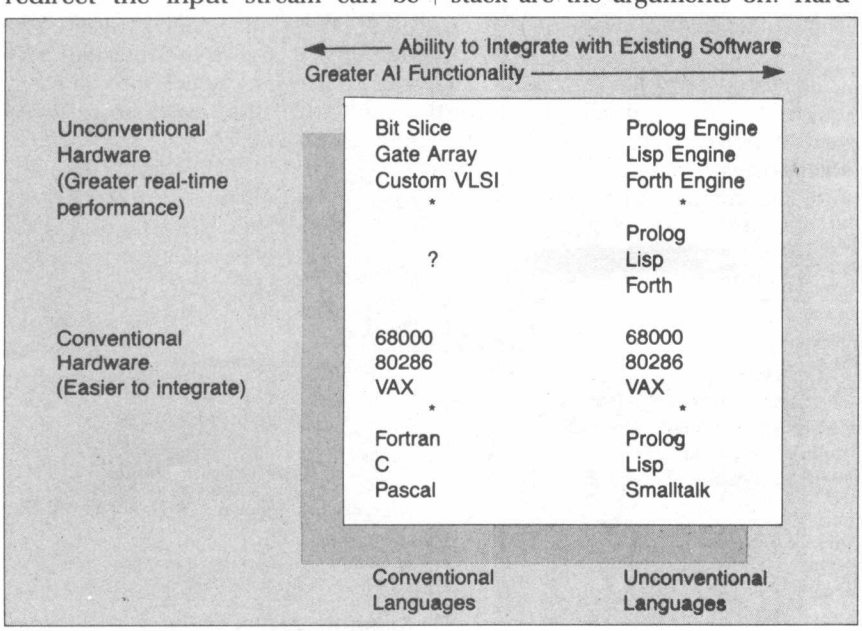


Figure 1: *Hardware and software choices*

(continued from page 91)

ware implementations favor the separate FP stack provided by the hardware. Software implementations favor the efficiency of having everything on the normal (data) stack. Judging by the discussions at the last ANS Forth meeting, the separate FP stack wins hands down. Still, you can have your cake and eat it, too. Imagine a definition of *F+* such as the following in a software FP system:

```
: F+  [FPOP]  (add numbers here)
                              [FPUSH] ;
```

In a Standard program, *[FPOP]* would pop an FP number from the FP stack and push it on the data stack. *[FPUSH]* would have the reverse action. Nonstandard programs that required maximum speed would simply redefine *[FPOP]* and *[FPUSH]* before compiling the same FP extension:

```
: [FPOP] ; IMMEDIATE
: [FPUSH] ; IMMEDIATE
```

The existence of a separate FP stack implies the existence of FP stack operators, and most FP packages supply *FDUP*, *FDROP*, *FSWAP*, *FOVER*, and *FROT*. In some packages, *F@* and *F!* move FP numbers to and from *FVARIABLE*s, and *FCONSTANT*s such as *PI* aid readability. FP numbers can be compared with *F<* and *F>*. *F=* may be provided, but comparing floating-point numbers for equality is a questionable practice and should be avoided. It is meaningful, however, to compare an FP number to 0 with *F0=* or *F0<*. *FABS*, *FNEGATE*, *FMAX*, and *FMIN* are usually also available.

How to input an FP (real) number varies from Forth to Forth. Most Forths install some kind of automatic conversion to FP when the FP extension is compiled. In polyFORTH, numbers containing a period and followed by at least one nonblank character are converted to FP—for example, *2.0E* would be a "real" 2 whereas *2.* would be a double-precision 2.

In MasterFORTH, UR/FORTH, and MacFORTH, all derivatives of the FVG Standard, real numbers must contain an uppercase *E*—*EE*, or *2.E*, or *2.0E*, or *2E0*, and so on.

You must be in *DECIMAL* or strange things happen. To print a real number, use *F.*, but first set the number of places to the right of the decimal with *PLACES*:

```
4 PLACES
3.14159E F.
```

prints 3.1416.

The polyFORTH *F.* is an exception to this rule and takes the number of places from the stack. If you are a polyFORTH user and would like compatibility with the FVG Standard, redefine *F.* in this way:

```
VARIABLE Places
: PLACES ( n ) Places ! ;

: F. ( r ) Places @ F. ;
```

Let's stop here for now. You can see that without much effort the floating-point packages of the various Forth vendors could be brought into alignment, and this is one of the things the ANS Forth committee is trying to do.

Now let's see what's involved in writing a floating-point package, in case you would like to experiment with it. First, you need a proper representation. The most efficient representation for Forth seems to be to use a normalized, double-precision signed mantissa and a single-precision signed exponent with the exponent on top. This gives you about nine digits of precision and an unheard of dynamic range. The high bit of the mantissa is the sign bit. The remaining bits form a normalized mantissa, unsigned, with the second highest bit set to 1. The binary point is to the left of this bit. This gives the mantissa 31 bits of precision. The number 1 would be:

```
HEX 0000 4000 1 DECIMAL
```

or "one-half times two to the one power."

To multiply and divide these essentially double-precision numbers, you're going to need some extended math operators, shown in Table 1, below.

Writing the various stack-manipulation and memory-access words is also straightforward: *FDUP*, *FDROP*, *FSWAP*, *FOVER*, *FROT*, *F@*, *F!*, *F,*, *FCONSTANT*, *FVARIABLE*, and *FLITERAL*.

Rather than support a fancy system of infinities and "not-a-numbers," the only special case I will handle is 0. A zero number will have a zero mantissa, which can be easily tested with the expression *OVER 0=*, as follows:

```
: QNORM ( q - t exp)
\ normalize q to bit 30;
\ leave adjustment as exp.
  2DUP OR 2OVER OR OR
  IF 1 ( count) >R
   BEGIN DUP 0< NOT
             WHILE Q2* R> 1- >R REPEAT
   2>R SWAP DROP 2R> TU2/ R>
  THEN ;

: ROUND ( t - ud exp )
\ assumes hi bit is zero.
  32768 0 0 T+ ROT DROP
  DUP 0< DUP IF >R DU2/ R> THEN ;

: F* ( r r2 - r3)
  f0= IF FSWAP THEN
  FOVER F0= IF FDROP EXIT THEN
  ( exp2 ) >R ROT ( exp ) >R  UNPACK >R
  2SWAP UNPACK >R DUM*
  QNORM ( t exp ) >R
  ROUND ( d exp ) R> + ROT ROT
  2R> XOR PACK ROT 2R> + + 1+ ;
```

Example 1: *Normalizing and rounding the quad-precision result*

Forth Word	Meaning
D>SHIFT (d u – d2)	Double-shift d] ['u bits to the right arithmetically
DU2/ (d – d2)	Unsigned double-precision right shift
T+ (tA tB – tC)	Add two triple numbers
T2* (t – t2)	Shift triple-precision number left once
TU2/ (t – ut)	Signed triple-precision number right shift
Q2* (q – q')	Shift quad-precision number left once
DUM* (ud1 ud2 – uqproduct)	Multiply unsigned double-numbers to yield unsigned result
DUM/ (uq ud – udquotient)	Divide unsigned quad uq by unigned dividend ud
DUM*/ (ud ud2 ud3 – ud4)	(ud * ud2)/ud3 with quad-precision intermediate

Table 1: *Extended math operators*

```
: f0= ( r - r f) OVER 0= ;
: F0= ( r - f) ROT 2DROP 0= ;
```

Changing the sign of the mantissa is easy, but you must not change the sign of 0:

```
: FNEGATE ( r – r2)
\ 2's complement of R.
   OVER IF >R [ HEX ] 8000
   [ DECIMAL ] XOR
      R> THEN ;

: FABS ( r – r2)
\ absolute value of R.
   >R [ HEX ] 7FFF [ DECIMAL ] AND
R> ;
```

Packing and unpacking the sign bit from the mantissa is also easy:

```
: UNPACK ( d – ud sign)
   DUP FABS ;

: PACK ( ud sign – d)
   [ HEX ] 8000 [ DECIMAL ] AND OR ;
```

Probably the simplest function to implement is *F** because exponents add and mantissas multiply. You do, however, need some way to normalize and round the quad-precision result, as shown in Example 1, page 94.

The other arithmetic primitives—*F+*, *F–*, and *F/*—follow a similar pattern. For a simple four-function package, you lack only the primitives for input and output, shown in Example 2, this page. *>F* converts a double to a real number, and *F.* prints it. This simple version of *F.* clips the real number to only a part of its dynamic range, limiting it to 999999999. Numbers less than 0.000000001 are printed as 0. A more sophisticated version would use logarithms to change the number from a power of 2 to a power of 10 before printing.

```
: F.  OVER >R FABS DUP 0>
  IF  0 0 ROT 0
    DO Q2* LOOP Q2* 999999999. DMIN
     2SWAP DU2/
  ELSE NEGATE D>SHIFT 0 0 2SWAP THEN
  500000000. 1073741824. DUM*/
  <# # # # # # # # # # #
  [ ASCII . ] LITERAL HOLD 2DROP
  #S R> 0< SIGN #> TYPE SPACE ;

: FLOAT ( d - r)
  2DUP D0= IF 0 EXIT THEN
  SWAP OVER DABS 1 ( count) >R
  BEGIN DUP 0< NOT
  WHILE D2* R> 1- >R REPEAT
  DU2/ ROT PACK R> 31 + ;

: >F ( d - fn)
\ converts most recent double number
\ to mixed fraction.
\ Used like 3.14159 >F
  DPL @ 0< ABORT" Needs dec point"
  FLOAT
  DPL @ ?DUP
  IF 1 0 ROT 0
    DO 10 0 DUM* 2DROP LOOP
    FLOAT F/
  THEN ;

  3.14159 FLOAT F. prints 3.141589999
  3.14159 FLOAT FCONSTANT PI
  PI PI F* F. prints 9.869587719
```

Example 2: *Input and output primitives*

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

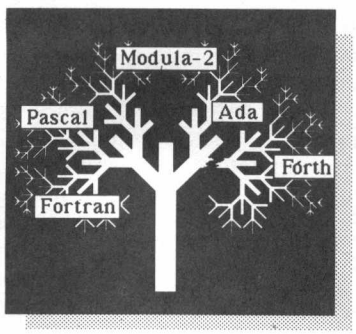
DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

COLUMNS

STRUCTURED PROGRAMMING

Mixed Reviews on a Bunch of 4.0s



by Kent Porter

A few months back, Microsoft invited a bunch of us computer press types to an all-day shindig so they could tell us about the high-performance languages. They even gave us all sweatshirts so we could prove we were there (I use mine to sleep in, subliminal messages being what they are).

During one of those sessions, a presenter made the interesting comment. "We had things in the works for Pascal until somebody else came along and took the market away" he said. Given the main topics of the seminar, it seemed clear at the time, that Microsoft had abandoned the PC-based Pascal market to this unnamed "somebody else." It also seemed clear that they had chosen to dig their trenches around C and Basic. But now things are not so "clear," because the biggest name in little computer software has released its own Pascal 4.0.

The version level numbers are pure coincidence. These numbers will be short-lived anyway, since Borland is about to introduce Turbo Pascal 5.0. Borland's version of Turbo Pascal will have a debugger and thus go head-to-head with Microsoft's Pascal 4.0 Codeview connection, which in turn will trigger some newer and better thing from Microsoft, and so on, and so forth. You know how that goes.

Anyway, this month's column examines the new Microsoft Pascal. Some good news and bad news will be shared along with some comparisons to Turbo Pascal 4.0. Then we'll talk about another 4.0, about which there's only bad news: the new LIM EMS standard.

Looking at MS-Pascal 4.0

The first thing that hits you about Microsoft Pascal 4.0 is the quantity

of the documentation—a 3-inch pile of paper. This documentation consists of the standard *User's Guide* and *Reference Guide* (little has changed, if anything at all, from the previous version) plus a *Version 4.0 Update* of 68 pages. Additionally, the documentation includes two Codeview manuals (the basic book and an update), plus a guide to the new Microsoft Editor.

That should give you an idea of the major enhancements in Microsoft Pascal 4.0: Codeview and the editor. Less obvious, but equally significant, are an upgrade to the large memory model (the only model now available), full-scale support for Windows development, and an OS/2 option.

One of the great disappointments about these documents is Appendix C of the *Reference Guide*, which discusses the differences between Microsoft and "other Pascals." What other Pascals? UCSD, for crying out loud. Give me a break! Who are these guys trying to kid? If Microsoft wants back a piece of the Pascal action, they won't get it by pretending that Turbo doesn't exist.

Another great disappointment about the manuals is that Microsoft still hasn't put related information all in one place nor indexed the information in an intuitive manner. Seldom can you simply look up something in the Microsoft Pascal docs; it's a sort of scavenger hunt with hints cleverly concealed by the indexer. The new manuals are no improvement over the old. In fact, the new documentation compounds the problem because you must now hunt through six manuals, compared to the earlier two. For example, it took me more than an hour to find out how to get a program into the symbolic format Codeview expects.

The compiler package runs on a DOS machine, but it can act as a cross-compiler for OS/2 development. An installation option lets you set up the system for OS/2 protected mode, OS/2 real mode plus DOS, or both. Link-time directives specify in which environment the executable runs. This is one area where Microsoft has a decided edge over Turbo, which hasn't yet heard of OS/2.

You can run this development system on a dual-floppy machine if you have masochistic tendencies. The manuals assume that you do, which I suppose is valid as a worst-case treatment. In that event, you'll have to swap diskettes in the A: drive three times to compile and link. A hard disk is so strongly recommended that it is almost mandatory.

Compilation is a two-step process with an optional third pass. The first two steps (PAS1 and PAS2) compile and optimize the source code into object form. The third step (called PAS3) produces an object code listing with numbered lines and a symbol table. The results are then linked to produce an .EXE file. The package comes with the latest segmented overlay linker (Version 5.01), plus an OS/2 linker called BIND.

Pass 1 of the compiler has a dozen command-line switches, each with a corresponding metacommand that can be embedded in the source file. In case of conflict between the command line and a metacommand, the metacommand wins. Most of these are pragmas for various check routines—subscript ranges, dereferencing of NIL pointers, stack overflow, and so forth. Some also pertain to the generation

of debugger information in the .EXE file. For a change, the user manual lists these metacommands on one page (in Table 5.2) and gives the defaults (all off). The other passes don't have switches, but the linker has nine. (All explained in Table 6.2.)

The 80X86 limits a segment size to 64K, which is the upward bound on any given code segment generated by the compiler/linker. The solution is to develop modular programs, compile them separately, and link them to form large applications. Each module then occupies its own code segment, thus allowing the program to transcend the 64K limit on individual segments. A routine in one segment can call a subprogram in another, so long as the external subroutine is declared PUBLIC in its owning module and EXTERN in the caller's; the result corresponds to a far proc in assembly language.

Microsoft Pascal's implementation of units is superior to that of Turbo Pascal 4.0. The Microsoft concept more closely parallels Modula-2 (from which it was borrowed). In Microsoft Pascal a unit consists of two separate files, the Interface and the Implementation, and a mechanism exists for explicitly exporting identifiers. In Turbo, a unit is one file that consists of both parts, and everything in the Interface section is implicitly exported. The Microsoft approach more readily allows the implementation to be hidden from the caller, which is desirable in group projects and commercial packages.

The data segment of a Microsoft Pascal program is also limited to 64K. This segment contains all global variables, memory resident constants, the default heap, and (gulp!) the stack. Including the stack in the data segment is a bad design decision on the part of Microsoft, because a heavily recursive program with a lot of data can easily run out of stack space and have a nervous breakdown. Turbo does it better, with an entirely separate stack segment that can be up to 64K.

On the other hand, Microsoft offers more flexibility with heap allocations. You can use the default (near) heap within the data segment, or the long heap located in all the uncommitted memory of the system. The only heap *in Turbo is the* long one, which can cause problems when one program spawns another. Microsoft provides a plethora of mechanisms for dealing with the two heaps.

At the aforementioned press briefing, Microsoft made a big deal out of its optimization, so I thought it might be instructive to test for it. One of the things Microsoft compilers purportedly do is to move invariant computations outside of loops. Here's a simplistic example:

```
FOR w := 1 to 100 DO BEGIN
    x := w + 10;      { loop-variant }
    y := 12;          { invariant }
    z := (y * 13) DIV 7; { invariant }
END;
```

In other words, *y* and *z* need not be recalculated in each loop, since their values don't deviate from one iteration to the next. Optimization

STRUCTURED PROGRAMMING
(continued from page 101)

is supposed to place these calculations outside the loop.

Well, guess what? Microsoft Pascal doesn't optimize for this condition, regardless what Microsoft says. I wrote a little program with this kind of construct inside the loop, and then moved it outside. Here are the results:

With invariant calculations inside: 0.98 seconds
With invariant calculations outside: 0.44 seconds

Turbo doesn't claim to optimize for this, and their results for the same program versions were 1.10 and 0.49 seconds, respectively.

While on the topic of performance comparisons, I translated three of the standard Berkeley benchmarks from C into Pascal and ran them under both Microsoft and Turbo. The benchmarks are as follows:

sieve Tests array indexing and integer arithmetic
fib Tests recursion
acker Tests recursion and integer arithmetic

Execution times for the benchmarks on an 8-MHz AT clone with no-wait state memory were as follows:

	Turbo	**Microsoft**
sieve	40.97	31.36
fib	22.68	24.34
acker	12.36	12.58

In other words, Microsoft does substantially better at sieve, and the other two benchmarks are somewhat of a wash. A comparison of code sizes reveals some startling differences:

	Turbo	**Microsoft**
sieve	2128	27,501
fib	2064	19,267
acker	2096	19,283

If the Microsoft compiler is so optimizing, why does it generate 10.5 times as much code on average to do the same job as the compiler from Borland?

STRUCTURED PROGRAMMING

It happens that I make a living from nine to five as a software engineer in a big Unix house. I know from personal experience that you can optimize a compiler to recognize the standard benchmarks and to perform superlatively when it thinks its results have a chance of being published. This is optimization for the wrong reasons, inasmuch as benchmarks are synthetic programs that don't necessarily parallel real-world programs.

Suspecting that Borland might be guilty as charged, I wrote another program outside the realm of the standard benches. This one is SINES.PAS, which took 1000 sines on my AT clone (without a math coprocessor). Here are the results:

	Time	**Size**
Microsoft	5.12	19,413
Turbo	2.60	3,248

The upshot of this exercise is that Borland's minimally optimizing compiler is either very smart indeed, or else it really does outperform Microsoft's by a 2-to-1 in timing and a 6-to-1 size ratio for a floating-point emulation application. So much for Microsoft's optimization.

Besides OS/2 support, the Microsoft compiler offers a couple of things that the Turbo compiler does not—Codeview and Windows. Codeview is a superlative debugger for .EXE files. It lets you set breakpoints and watchpoints, step through source lines and see where a program is going awry because of a runaway loop or whatever, monitor the stack, and so on. Maybe I'm old-fashioned; I still prefer to stick WRITELN statements into a misbehaving program to find out what's going on, but I make an exception where it comes to Codeview. It's great, and the ability to control debugging through three different means makes it the most intuitive debugger around. One of these days, we'll stack Codeview up against the soon-to-be announced Borland debugger and see how they compare. Meanwhile, Codeview wins the day.

Microsoft claims that its C and Pascal 4.0 are the only languages that support Windows. That's not entirely true. For example, Actor is an object-oriented language for AI and Windows, which is Microsoft-approved. Although Actor is not a mainstream language, it does negates the assertion. Certainly the mainstream Turbo languages don't work with Windows; I struggled to write a Windows application in Turbo C and the program just wouldn't compile, despite help from Borland.

You could make the waggish observation that Microsoft's near-sole support is a commentary on the acceptance (or rather lack thereof) of Windows within the programming community. Nevertheless, the advent of OS/2 Presentation Manager—a glorified Windows—makes this unwieldy software the user interface of the future. For those who want to get a head start on that future, Microsoft Pascal 4.0 is an attractive alternative to the more arcane C language. Microsoft Pascal is the native language of Windows,

which enforces the Pascal calling convention on C programmers who want to use its interface. Version 4.0 provides full access to the richness of the Windows environment by means of the WINDOWS.INC include file and the $WINDOWS metacommand.

Microsoft Pascal is purely a high-level language. Some of the intrinsic functions and procedures coat specific DOS calls with syntactic sugar. The language does lack general-purpose, low-level calls comparable to Turbo's MsDos and Intr procedures. Also, no direct access is provided to registers from Microsoft Pascal. If you want to do low-level stuff such as generating software interrupts, you have to write subprograms in assembly language and link them as externals with the Pascal program. This makes it necessary for serious developers to have an assembler (and guess who just happens to sell the best-known of them). It also complicates the development process. Microsoft hasn't been reluctant to extend the language in other areas, and they would have done their Pascal users a great service by providing low-level calls and register access directly from the source level.

On the other hand, none of the major software manufacturers has its mixed language act together as well as Microsoft. You can link Pascal modules not only with assembly-language routines, but also with C and Fortran modules. Just as Microsoft C has an attribute for specifying the Pascal calling convention, Pascal also has a C-convention option. These modifiers govern the order in which parameters are physically passed to subprograms. Also, Pascal offers a VARYING attribute that allows Pascal programs to imitate C in passing a variable number of arguments to a subroutine.

Microsoft also has the advantage over Turbo in the areas of object portability and linkage. The Turbo linker is indistinguishable from the compiler and limited because it allows only the importing of assembly-language routines and the most rudimentary C routines. You can't export a Turbo Pascal object module because no such thing exists. Also, Turbo doesn't support any overlays. The Microsoft linker is a separate program altogether, supporting mix-and-match of object modules coming from various languages and furnishing extensive support of overlays. Thus you can create programs in which each module is written in the language best suited for its task; Fortran for computations, assembler for low-level access and speed; Pascal for record processing; and C for control. The linker then joins them into a program in which little-used sections can operate as overlays. Pretty impressive stuff.

Also impressive is the new Microsoft editor, a welcome addition to the programmer's toolkit, which comes with Pascal 4.0. It's a windowing environment (though not a Windows application) that allows you to view and work on multiple source files, or different parts of the same file, in bordered text windows. The editor includes all the features you'd expect from an advanced program development tool. In effect, the editor creates a work environment much like Quick and Turbo, but more extensive because of windowing, a clipboard, and word-processing-like operations. It includes compile-link-and-go without leaving the editor.

In default mode the editor bears a visual and operational resemblance to Brief. The Microsoft editor is infinitely customizable with macros and provides the ability to assign the macros to keys. You can also write C routines and install them as editor functions. A control file called TOOLS.INI stores your configuration as ASCII text strings, which you can edit at will. The editor also comes with preconfigured files that make it emulate the Brief and Epsilon editors, the Quick environment, and Wordstar.

Will the new Pascal release win back the market for Microsoft? Probably not. It's harder to use than Turbo, doesn't offer as many extensions, takes much longer from source to executable, produces .EXE files that are unnecessarily large, and costs three times as much. For medium to light-heavyweight development, the only thing I can think of that makes Microsoft Pascal 4.0 preferable to Turbo 4.0 is Codeview.

On the other hand, Microsoft has now elevated Pascal into the same league with its venerated C and Fortran compilers for true heavyweight systems development. Especially strong are its support for Windows, OS/2, and mixed-language programming. These features are bound to win back some share of the Pascal market and to attract others who find C too esoteric for their tastes.

And Now For LIM 4.0

I had a terrific idea for this month's column. Four months after I ordered it on an emergency basis, Intel finally sent my LIM 4.0 upgrade kit. I thought I'd write a column about using the new EMS standard for interprocess communication. This made sense, since the focus of this month's issue is real-time programming, which implies multiple processes, that often need to communicate among themselves, right? Well, guess what. The upgrade kit didn't work.

My problems started with the first page of the slender leaflet. It said: trust me, all your problems are solved if you just run the INSTALL program. Uh-huh. Half a dozen panels into the program, the instructions tell you to please wait for the one second to several minutes required to find out how much memory you have. It was late, so I let the program survey all the world's memory while I showered. Then I rebooted, restarted INSTALL, and waited a while longer. Then I rerebooted and re-restarted INSTALL, after that I went to bed. In the morning, it was still trying to quantify my paltry 1 Mbyte of EMS.

Since that failed, I tried to understand the gobbledygook on pages 3 through 10 of the upgrade instructions. The instructions purport to tell you how to manually install the upgrade. Trouble is, it's written in tongues. I speak nine languages fluently, and one of them is computerese, but my eyes glazed by page 4. In short, I couldn't get LIM 4.0 up and running, so I reviewed Microsoft Pascal 4.0 instead.

I griped about the LIM 4.0 problem among my computer buddies, and they all clucked and shook their *heads*. "*A bunch* of junk" was the consensus. Then Ron Copeland of *DDJ* relayed a rumor to the effect that LIM 4.0 is incompatible with the VGA, which I have on my AT. I discussed it over lunch with Tyler Sperry, who was still trying to recover from my original vitriolic attack on Intel. At his suggestion, I pulled my VGA and reinstalled the EGA from which I had upgraded shortly before.

Hey, suddenly it worked! Intel's INSTALL ran without a hitch. Whoopee, now I have more EMS capability and a lesser graphics board. Is the trade-off worth it? I don't think so. *I shouldn't* have to make that trade-off, nor should you.

It's no secret. If you look at the "Examining Room" column in this magazine, or in numerous other magazines, you will notice that I do a lot of product reviews. I don't like to be tough on little guys. They labor hard in their garages and there's much blood, sweat, and hope in what they produce. I have empathy with the little guys; I'm one of 'em.

A lot of us are.

But Intel isn't a bunch of guys in a garage. They're a big outfit, and they ought to know better than to release this kind of flawed stuff. Months late, I might add.

I know how EMS works. I just wrote a chapter on it for my forthcoming book on advanced programming in Turbo C. EMS needs a 64K frame buffer in high memory, and it just so happens that VGA hogs a lot of memory exactly where EMS wants it.

So what we have here is a conflict of standards: VGA vs EMS 4.0. That's not my fault and I resent—on behalf of us all—having to make a choice between better memory management and better graphics. The way VGA works is hardly proprietary information. The two standards converge on one vendor, whose initials are IBM, which owns a substantial investment in Intel, who is the object of this diatribe.

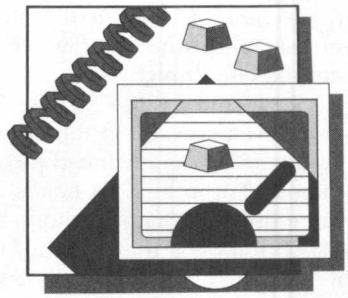
One could make a case that EMS was here first, and thus owns a legitimate stake in some unclaimed piece of high memory. On the other hand, VGA is hardware, which is by definition inflexible, whereas the EMS frame buffer is determined by a software device driver. The software ought to be smart enough to figure out how to coexist with VGA.

That it isn't is a discredit to all those Big Guys who allegedly have our best interests at heart. They screwed up, plain and simple, and it's you and I who have to pay the price for their inattention.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

SD '88, Prolog Tools, and Transputers



by Michael Swaine

There was an edge to some of the audience's questions. It didn't faze the panel of experts.

More than one member of the audience for the panel discussion on software development in the 1990s at Miller Freeman's Software Development '88 conference (SD '88) asked essentially the same question: "Today, the emphasis in object-oriented programming seems to be on software development as a creative activity. Can't we get a little more scientific, can't we move toward an engineering approach to object-oriented programming? More specifically, are there any rules or strategies you can point to that will help us decide what objects to define?"

The questioners got little satisfaction. The panelists' answers ran thus: Chuck Duff, author of the object-oriented programming language Actor: "A good plan is to study the physical system you are trying to model and create the classes of objects it has."

Dick Gabriel of Lucid, deeply involved in object-oriented Common Lisp: "That's a fundamental question for which there is no easy answer. I try things."

Bjarne Stroustrup, author of object-oriented C++: "It's a holy grail. There is no panacea."

Chuck Moore, author of the Forth programming language, which doesn't have to be object-oriented: "Programming is an art; we might hope it becomes a craft; it will never be a science."

The panel members represented several programming paradigms besides the object-oriented one. Although there are object-oriented *Forths and Lisps*, those languages are surely paradigms unto themselves; and moderator Stan Kelly-Bootle was there to represent the procedural paradigm single-handedly if he had to. (He's written extensively on Modula-2, has just finished writing a book on C, and was wearing an Ada T-shirt.) But with their combined knowledge of the object-oriented paradigm, one of them should have been able to field the insistent question if anyone could.

Maybe no one can. Maybe there is an invariant principle stating that, for any given paradigm and for any given state of the programming art (excuse me, discipline), there are aspects of the programming process that can be formalized and other aspects that can't. If so, surely a great deal of programmer effort has to be directed toward the latter. That's where only creativity will suffice. Maybe, as the panelists seemed to be saying, the decision regarding what objects to create when developing an object-oriented system is such an aspect.

Maybe. And maybe when you cross paradigm boundaries, the aspects shift. Then the first disorienting puzzle presented to you by a new paradigm would be to identify the problems for which only creativity will suffice.

On the Paradigms Beat at SD '88

Some of you may have attended SD '88. Ron and Jon and Allen and Tyler and I were there. SD '88 has grown in three years to become a truly important and informative conference for software developers. Of course, some of the sessions were less important and informative than others, and unless you knew the speaker, it was a turkey shoot. The networking opportunities were good, though, as was the chance to see people whose work you've read. I had never seen Bjarne Stroustrup before.

I hope the conference sponsors can find ways to increase the conference's networking value next year. In addition to the sessions, the conference had exhibit space for companies, the main value of which was probably informal recruiting. Plans are for a greatly expanded exhibit program next year, and if that takes the direction of a sort of job fair it could be interesting.

This year there were tracks of lectures and workshops on artificial intelligence, database design, the C language, design methodologies, languages, and graphics. For the paradigmologist there was much to record. I attended many of the sessions I mention here, but since sessions ran concurrently I couldn't attend everything I was interested in. I'm summarizing some of the sessions from the proceedings.

While the speakers in the panel discussion I mentioned earlier weren't able to provide an engineering approach to deciding what object to develop in an object-oriented design, several sessions did deal with practical object-oriented programming issues.

The Oh-Oh Factor

Satish Thatte of TI talked about object-oriented database systems. OODB, Thatte argued, is a necessary step making smart front ends truly viable; conventional database architectures with AI front ends grafted on are handicapped by inflexibility. Citing the ten years it took relational database technology to be accepted commercially, he predicted that it will take OODB five to ten years to reach the market.

OODB represents a significant para-

digm shift for database developers. Object-oriented programming may be the paradigm shift challenging the largest number of programmers today.

Chuck Duff and Mark Solinski led a workshop on Actor development. Along with the developers of Smalltalk, Chuck has the distinction of having developed a commercially successful language strictly for object-oriented programming. Actor is a pure object-oriented language, down to the activation records on the stack (they're objects, too). I missed his talk, so I called him after the show and he gave me a little more insight into object-oriented programming in general and Actor specifically.

Chatting with Chuck

Chuck talked about multiple inheritance, the ability of an object to inherit from more than one parent, and about why he left it out of Actor and has no plans to add it. "At the implementation level," he said, "it turns simple tree traversal into arbitrary graph traversal. Somehow you have to linearize the graph. Smalltalk-80 did it by copying code, physically copying the methods." Duff called this a cop-out. Linearizing the graph is not impossible. "You can unfold the graph. But it adds code bulk," he said.

At the user level, Duff sees another kind of problem. He fears that users will view multiple inheritance as a panacea and misuse it. It is difficult to avoid conflicts, and the efforts necessary to do so may "make you wonder if you are really simplifying anything." He acknowledged that Lisp systems such as Flavors have implemented multiple inheritance, but he says he's seen some unreadable Flavors code result from that decision.

Having published last month some of Bjarne Stroustrup's views on what makes a language object-oriented, I asked Duff to talk about the defining elements of the object-oriented paradigm. "I think dynamic binding is fairly essential," he said. The Actor compiler works hard to convert dynamic bindings to static for efficiency, but the ability to use dynamic bindings supports what Duff calls "experimental programming." "Inheritance is necessary for reusing code. Ada packages are flat [do not support inheritance]," he added. Of course, the Ada people might disagree about the importance of inheritance, but he thought it fairly essential. "Encapsulation is widely accepted." Encapsulation, in conjunction with inheritance and dynamic binding, he said, is very powerful.

Returning to the question that had nagged the panel, he said, "We can be more scientific about it. We teach a course, and teach people to start with the physical system. The way its objects have evolved is probably a good way [to begin]." When the physical model is in need of redesign, he recommends doing systems analysis to find a better segmentation of the problem.

There will be more help for the object-oriented software engineer at this year's OOPSLA conference in San Diego in September, he said.

Back to SD '88: Another speaker, Rick Potter, discussed structured design for object-oriented programmers, pointing to a partial answer to the question that led off this column, but only a partial one. Yes, we can develop measures of goodness for objects and their interaction. No, that won't tell you what objects and classes to create.

What the Gods Would Destroy They First Submit to an IEEE Standards Committee

Object-oriented C was covered at SD '88 from several directions.

Lawrence Rosler predicted that the C programming language of the next generation will abandon at least one of the features that made the language popular: its compactness. C will get big, and will encompass alternative programming paradigms, certainly including object-oriented programming.

Bjarne Stroustrup gave an overview of C++, the proposed successor to C. He listed some of the features he left out of C++, including garbage collection, multiple inheritance (but AT&T has plans to add this), support for concurrency, exceptions, parameterized classes, and integration of the language with a programming environment. The benefits of these constraints, he explained, were compatibility, internal consistency, and efficiency. There were other workshops and lectures on C++ and Objective C.

Parallel Tracks

I found three talks that dealt with issues of parallelism.

Robert Ward had played around on the parallel machines at the Advanced Computing Research Facility at Argonne Labs and talked about programming large-scale parallel architectures such as Encore, Cray, and Hypercube. The focus of his talk was on shared-memory implementations, not communicating processes (although he claimed that the approaches are in some sense duals of one another, and that you can simulate one approach with the other). He argued the case for extending C to handle this sort of parallelism: the machines he was discussing all supported some form of Unix, which favors C, and C would make the programs more portable.

Since most parallel-processing work is in the experimental stage or is done for research projects where the developers don't see much need for portability, he had to justify this approach. Portability and maintainability are linked, he pointed out; also, the architectures are not stable. Moreover, developing a portable approach to parallel processing will facilitate the development of benchmarks for evaluating parallel architectures.

Finally, he answered the objection that machine specificity is necessary to get the performance benefits of parallelism. There are algorithmic benefits accruing from the use of a parallel approach, he said, but the benefits will be masked by fiddling with machine-dependent optimizations.

Mark Gluck and David Parker spoke cogently on neural networks, Gluck explaining why cognitive psychologists and neurophysiologists care about the stuff, and Parker sketching an algorithm.

Gluck argued that these models are more appropriately called parallel-associative networks since they

are in some ways not very neurallike at all. He sketched a brief history of associative net models, starting with the perceptron model of the early 1960s, which was unable to handle exclusive-OR; he told how Minsky and Papert shot down this model and everyone more or less abandoned the approach for a decade, and how it recently resurfaced when new algorithms were developed that implemented multilayer nets that do handle exclusive-OR.

Gluck has developed, with psychologist Gordon Bower, a model that accurately predicts human decision making in a medical prediction setting where the disease to be predicted is rare. Working with neurophysiologist Richard Thompson, he has applied a neural net model to sea slug neuron firing, with enlightening results.

Parker discussed an algorithm he developed for neural nets. He summarized the neural net approach succinctly. All learning is minimization, he said, generally minimization of error, and we have many good algorithms for minimization. The neural net approach is nothing but the parallelization of a minimization algorithm. His own algorithm is a parallel version of the steepest descent algorithm.

He pointed out that there was absolutely no performance advantage to the neural net approach over sequential minimization without parallel hardware. There may be design advantages, though.

Avram Tetewsky talked about tasking, the Ada facility for concurrent programming based on Tony Hoare's CSP language. He warned against the use of the simple task construct, in part because it limits your flexibility in passing data between tasks.

I Didn't Say That

Several speakers presented what one would normally think of as AI languages, and many of these speakers concentrated on non-AI uses of the languages. It looked as though the conference organizers had asked the speakers to demonstrate that AI tools could really be used for serious purposes.

Dick Gabriel talked about Lisp as a general development language and as a systems language. He showed how to develop a sort of generalized spreadsheet using the Common Lisp Object System.

It was Gabriel, incidentally, who loudly ridiculed Sun's plan to rewrite Unix in C++. Structured programming, he claimed, had threatened to stop cold the pace of advancement in software development in the mid-1970s, but, as it turned out, only retarded it for five years. C and Unix, he said, will stop us cold for 25 years. It was Dick Gabriel who said that, remember—not me. I'm just an innocent paradigmologist. Gabriel's at Lucid Inc., in Cambridge.

John Malpas presented Prolog in one workshop as an application language and in another in a software engineering context. He pointed out that the self-descriptive quality of the language makes it possible for a Prolog program to document itself to some extent.

To find out about SD '89, write to Miller Freeman, Seminar Dept., 500 Howard St., San Francisco CA 94105, or call 415-397-1881.

How Logical Is Prolog?

You know John Malpas's work: He did an article for us on Prolog. He and Dave Cortesi and I have made the most fuss over Prolog in these pages, and I wonder if I shouldn't feel a bit guilty about my part. There are a lot of people playing with Prolog today, and I choose that verb deliberately.

In a previous life, I was a consultant in research design and data analysis. It troubled me that many of my clients, all graduate students and faculty members, wanted to perform statistical analyses whose assumptions they did not understand. Now, as someone who has encouraged the widespread use of Prolog, I must take some of the guilt for the legions of Prolog programmers who don't know what resolution is.

To relieve my guilt, I'll tell you about a new book on Prolog that just came in. The book is *Prolog Programming in Depth* by Michael Covington, Donald Nute, and Andre Vellino (Glenview, Ill.: Scott, Foresman, 1988).

About half this book is spent defining the language, which the authors do well and at a level an experienced software developer can appreciate. The discussion is strong on practical tips and bibliographic references, and on how features have been implemented in different compilers. There are also appendixes on debugging and on features of Arity's and Borland's Prolog products.

The other half of the book presents artificial intelligence applications. There are no surprises in the selection of topics—search heuristics, expert systems, inference engines, natural language processing—or in the example programs the authors include.

Never Tell Me the Odds

The book is informed and informative. For example, the authors raise doubts about the confidence factors widely used in expert systems. Abundant research shows that people, expert or not, are poor at assessing conditional probabilities, and in fact at assigning numbers to just about anything. Confidence factors ought to be examined with a skeptical eye, and these authors are appropriately skeptical.

They spend just one chapter laying the logical foundations of Prolog, but they deal with the implications of its logic throughout the book.

They do explain resolution and how Prolog uses resolution, producing proof trees via SLD resolution. They explain that SLD is sound (never letting you infer something that doesn't logically follow from other statements) and complete (finding all possible inferences), and explain how Prolog's implementation of SLD resolution is sound but not complete, and they tell why it was implemented that way. They talk about the closed-world assumption and the way Prolog handles negation, and what these things imply.

Nevertheless, I wish the authors had said more.

They give practical advice on the use of the cut operator, but don't fully clarify the effects of cut, which some people fear can compromise the logic of a program. They should have said that the cut operator has no logical significance whatsoever: Its use cannot change the logic of a Prolog program. Cut just prunes the

proof tree, with a gain in efficiency but a loss in completeness.

What the authors refer to as "red" cuts are a special case. Here the programmer consciously writes code that is declaratively incorrect (logically incorrect), depending on his knowledge of the order of clause evaluation to keep the program from crashing. I wish the authors had come down harder on this kind of programming, which undermines any notion of Prolog as programming in logic, and ties the code to nonparallel implementations.

The principle of negation-as-failure and the closed-world assumption (CWA), both relevant to the logic of Prolog, are not equivalent. Frankly, I can't tell you how they differ, although I know that CWA is more powerful. But unless I missed it, the authors of this book don't clarify the point, and I think it is worth discussing in a book that examines Prolog programming in depth.

More Books

Despite these points, *Prolog Programming in Depth* is a good book. For the time being, though, the Prolog programmer who really wants to understand the logical structure of the language he or she is using may just need to read a book on the relevant aspects of logic. One on my shelf is *Foundations of Logic Programming* by J.W. Lloyd (New York: Springer-Verlag, 1987).

For the less committed, two good books that explain resolution briefly (in a chapter or appendix) are *Mathematical Theory of Computation* by Zohar Manna (New York: McGraw-Hill, 1974), an older book with 20 solid pages on resolution; and *Natural Language Understanding* by James Allen (Menlo Park, Calif.: Benjamin/Cummings, 1987). You would not buy either book just for their treatment of resolution, but both are books I thought you might have access to, and the Allen book is worth getting if you have any interest at all in natural language processing. I recommend them because I don't think it's reasonable to expect people who are merely experimenting with Prolog to buy and digest a book on mathematical logic. And most of the purchasers of Prolog products today bought them precisely to experiment with the language.

Transputer Meditation

With any new paradigm, the first thing you want to do is experiment with it; see what its model problems look like and where it demands creative thinking, or creative rethinking about familiar problems.

Parallel processing is a radical paradigm shift, encompassing not just one class of new paradigms but a whole curriculum of them. The most radical of these can force you to rethink fundamentally how you approach familiar problems. I opened the topic of parallel processing here last month, talking about the INMOS transputer chip, and about occam, the language developed for programming transputers. I thought I had said about all I could until I actually got my hands on a transputer development system to play with.

But after I wrote that column, my Munich-based editor friend Jürgen Fey dropped by. He was in the country on a quick trip to SD '88 and other Silicon Valley attractions, and, as he usually does when he comes to California, he found time for a visit. We ate lasagna and drank California wine, swapped stories and the names of some good books, played with the dog and bounced on the trampoline, and finally, around midnight, we sat down and talked transputers.

Jürgen had been able to get his hands on a transputer development system to play with, and had been building a transputer board. He reminded me that, in the time since INMOS had designed the transputer chip to be used for parallel processing, a number of system development projects had been using transputers, proving the chip's practicality. Many of the projects were defense industry jobs, where details can be hard to come by and cost considerations differ from those in commercial markets. Nevertheless, such companies as Sun and Atari are now investing in transputers for commercial applications. Jürgen had been bitten, too. He was eager to get back to Munich to finish his transputer board to show at the CEBIT show in March.

The Parts of TDS

I asked Jürgen if he was doing his development using TDS, the development system supplied by INMOS, and what he thought of it. He said he was, and that it was solid. TDS includes occam, a linker, an editor, a debugger, libraries, and a configurer.

It's the configurer, in part, that would allow Jürgen to develop parallel-processing software on a single-transputer system if he wanted to. The configurer allows you to do your development work using one transputer, simulating a network of transputers in software, and then configure the program for a multiple-transputer system. You tell the configurer how many processors you have and where the links are, and that, I gather, is pretty much that.

Jürgen's initial system, though, contains two transputers, and that's because of the debugger. It's called a network debugger, and is particularly interesting, actually requiring a two-transputer system. The target program runs on one transputer, the debugger on the other. Jürgen says it's very powerful.

The folding editor is also interesting. It allows you to collapse detail, much as an outline processor does.

Jürgen then briefed me on the chip. There are three families of transputers now: the 16/32-bit T2xx, the 32-bit T4xx, and the 32-bit T8xx with an FPU. A transputer has four I/O ports called links, which facilitate the development of transputer networks. The occam language supports the links directly via what it calls channels.

Jürgen thinks the transputer is well-designed for parallel processing. In addition to the external parallelism it facilitates, there is a fair amount of parallelism inside the chip. Each of the four transputer links has DMA, and can perform memory accesses in parallel with each of the others and in parallel with the CPU, the FPU (if present), the ALU, and the integer unit.

Occam's Praiser?

Because of the transputer architecture and the nice match between the architecture and the occam lan-

guage, many things you would like to be able to do with parallel processors are easy. Jürgen drew quick sketches showing how you would implement multiplexors and systolic arrays with transputers.

Some things, though, are not so easy. Occam is not a rich language, and C and Pascal programmers will find some of its limitations annoying. Its inability to do mixed-mode arithmetic, for example, is annoying. Its lack of operator precedence is alarming. And together these limitations can produce code that is full of parentheses and explicit type conversions.

Although occam is a high-level language, it permits some assemblylike optimizations. One of the most important lessons Jürgen learned was that indexes must be kept internal to the chip and large arrays external. Since the transputer may have 2K to 4K of internal RAM, this can become an issue.

But when you get beyond the optimization tricks, parallel processing in any language can be a nightmare. Systolic arrays are a simple technique for implementing parallelism, but most of the parallel equivalents of sequential techniques are yet to be discovered. And Jürgen posed the question, how do you document a parallel-processing system for your boss/client? Nassi-Schneiderman diagrams won't work.

The Homebrew Computer Club vs. Japan, Inc.

One broad class of models that Jürgen thinks may prove fruitful is neural networks. Although the neural net model probably won't fit every parallel-processing problem, it is strictly parallel, and it works. Jürgen's next step will be to investigate neural net models. He thinks there are about 50 of them, and he'd like to get comfortable with at least 10 before he draws any conclusions about their usefulness for his goals.

Part of the appeal of parallel processing for me is that it forces you to jettison so much mental baggage. Jürgen, who also likes to travel light, likened the situation in parallel processing today to that of the Homebrew Computer Club in the 1970s, when hobbyists brought together their wire-wrapped boards and code and swapped ideas while hacking a trail to a new technology.

It's appealing to view parallel processing as a kind of hacker frontier, and that's not altogether wrong; but companies and governments with lots of money to spend have also been investigating parallel processing. Jürgen told of interviewing the head of Japan's ICOT, who talked about the Japanese commitment to research in parallel processing, and about their plan to develop an automatic parallelizer. The program would automatically convert any sequential algorithm to an efficient parallel form. The plan failed; the researchers had to settle for a simpler goal: developing a tool that would interact with a savvy programmer to help him parallelize the algorithm.

Jürgen said he took comfort in that failure.

Then Was Now and Now Is Then

Although I think it's clever, the above subhead doesn't really fit this closing note. But "then was now and now is then" has been haunting me, and I knew that if I didn't use it soon somewhere in my writing, it would insert itself into my conversation in some even less relevant way, probably making me look like a fool. Looking like a fool in print is something every writer gets used to. I hereby place "then was now and now is then" in the public domain: feel free to use it as you dare. You may even find an appropriate use for it, and then you won't look like a fool.

Five years ago, writing in *IEEE Spectrum*, Robert Kahn of DARPA gave this projection for computing in the 1990s:

Computer hardware: advanced packaging and interconnection techniques, ultra large-scale integration, parallel architectures, 3-D integrated circuit design, gallium arsenide and Josephson junction technology, optical components.

Computer software: concurrent languages, functional programming, symbolic processing (natural languages, vision, speech recognition, planning).

Computer performance: one giga-instruction per second to one tera-instruction per second.

Kahn was describing the fifth-generation computer technology, which the Japanese began planning for in 1979 and are pursuing with single-minded dedication today. I don't mean to hint that Kahn's predictions make him look like a fool. He may have missed on a couple of points, but some rough beast does seem to be forming out of the materials he inventoried, and the hour for some sort of fifth-generation computer technology's hour seems nearly at hand.

But whither does it slouch? Says Dick Gabriel: "Europe will be pouring six times as much government money into programming as the U.S. in the next decade. I expect the lead in software to move abroad."

Five years ago, it seemed plausible that the next generation of computer technology would be developed first in the United States. Today, based on funding and directness of effort, the most likely developer for fifth-generation computer systems is Japan, followed by a combined European effort, followed by the United States.

I guess it's a good thing we got all that practice reading their manuals when we bought their stereo systems.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 7**

EXAMINING ROOM

Peabody For Turbo C

Product:
Peabody for Turbo C
Target:
IBM PC XT, PC AT, PS/2, and compatibles
Requires:
Hard disk, DOS 2.1 or later, 640K recommended
Pricing:
$100
Vendor:
Copia International Ltd., 1964 Richton Dr., Wheaton, IL 60187; 312-665-9830

Peabody is one of those programmer's tools that, five minutes after you start exploring it, you wonder how you ever lived without it. A direct competitor to the Norton Guides, Peabody is an online language database for programmers. Peabody also includes some other reference materials and utilities to *help developers*.

Like the Norton Guides, Peabody comes in various language flavors. The one I used for this review was Turbo C. There are also references for Microsoft and Lattice C, Turbo Pascal 4.0, and DOS.

Peabody is intended chiefly to function as a TSR. You could run it as a standalone application, though I don't know why you would except for familiarization. Additionally, there's a mode called "tandem" in which Peabody becomes a temporary TSR. For tandem mode, you type a command such as *PEABODY TC*. This brings up Peabody in standalone mode, but Peabody starts Turbo C as a child process and then hovers in the background pretending to be a TSR. Similarly, you can run Peabody in tandem with Brief using the command *PEABODY B filename.ext*. An exit from the child also ends Peabody and removes it from memory.

This is an attractive feature, because Peabody takes a lot of memory for a TSR: 115,120 bytes. That's 60 percent more than the Norton Guides consume. On the other hand, Norton is a pure TSR and doesn't have a tandem mode; it's either resident or it's not. Peabody's tandem approach makes more sense, since you probably only want to activate the reference system while actually programming, and any other time the resident software wastes memory that could be used for other things.

The disk space requirements for Peabody and the Norton Guides are about the same, with both taking around 700K. The Peabody reference database is a little bigger: 542K versus 516K for Norton.

Peabody uses a minimum of four hot keys. For the Turbo C version:

- Ctrl-Tab brings up the database table of contents
- Alt-LShift serves as the Hyper-key
- LShift-Tab redisplays the most recent frame
- Ctrl-Backspace tags the current Peabody window as a "sticky frame"

These default hot keys are reassignable with a configuration utility called PBSETUP. Each additional Peabody database you install brings along its own default Hyperkey; Turbo Pascal, for example, uses LShift-Ctrl.

Hyperkeys and sticky frames are features that programmers are sure to love (although it sure would be

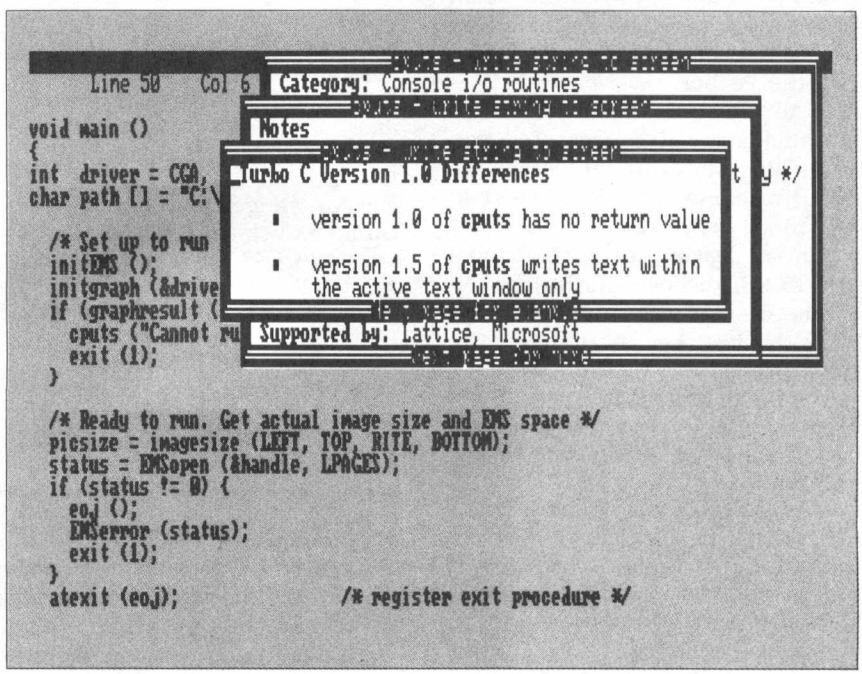


Figure 1: *Field Validation Options for PC/Forms*

Ron Copeland, associate editor for DDJ, is the coordinator for this review section. He welcomes your feedback on products worth reviewing.

(continued from page 124)

swell if Peabody included cut-and-paste, as well). The Hyperkey performs an automatic lookup of the language keyword at the current cursor position. For example, say you've forgotten some of the details of the Turbo C *cputs( )* function. You position the cursor on *cputs( )* in the source listing, then press Alt LShift, and shazam! Peabody opens a frame explaining *cputs( )*.

Norton does the same thing without a special hot key by automatically positioning the expand menu at the keyword indicated by the text cursor.

The Peabody frame is thoughtfully located where it won't overlay the keyword you're worried about. Successive presses of Enter bring up a stack of frames discussing general features, implementation- or version-specific issues, and a short program example. That's what Figure 1, previous page, shows. You can go backward through the stack by pressing Esc and forward again with Enter, removing and adding frames with single keystrokes. Sure beats turning pages in a manual.

A sticky frame is one that remains on screen after you return to edit mode; Norton has no equivalent. When the frame you want to retain is on top of the stack, you press Ctrl-backspace to tag it, then deactivate the Peabody session with Ctrl-Esc. All the Peabody frames except the sticky one disappear. You can move the sticky frame elsewhere with the cursor keys and revert to edit mode with Esc. This is tantamount to leaving an open manual next to the keyboard for further reference as you write the code. Any Peabody hot key evaporates the sticky frame.

The overlapping frames are a mixed blessing. Peabody's use of frames takes less total real estate per unit of information than Norton's quarter- to full-screen panels. That's necessary to implement sticky frames, and it leaves more of your source code visible. On the other hand, the small frames crowd and fragment the information. Norton gives you most or all of the information at a glance in a single, relatively uncluttered window. Overall, this makes Norton more visually appealing, but the ability to hang on to and move sticky frames as you edit your source code gives Peabody a distinctive advantage.

From the table of contents level (Ctrl-Tab), Peabody furnishes a hierarchy of menus that successively narrow down to the item you want to look up. You can get into the database by subject or keyword, and also by library functions, operators, data types, ASCII characters, and other categories. Norton provides similar paths, but by the alternative means of pull-down menus and cross-references. The outcome is largely the same, but the methods differ. Neither approach seems clearly superior; they're simply different ways of doing the same thing.

Peabody offers a useful utility that lets you examine memory or a file from within the environment in standard dump format. You can also view a directory, which is handy if you're using an editor that doesn't have a temporary exit to the DOS shell.

The content of the Peabody reference database for Turbo C seems reasonably complete. The only thing that's missing is the graphics subsystem introduced with Turbo C 1.5. This is a curious omission inasmuch as Peabody does furnish information about the text extensions in 1.5. I found no factual errors in the lookup material, whereas I did in Norton; that doesn't mean that Peabody has no mistakes, but just that I didn't notice them if they do exist.

In general, Peabody is an extremely useful, well-rounded programmer's aid that deserves a strong recommendation. For any serious programmer, it will quickly pay for itself in the productivity gains that come from not having to take your hands off the keyboard to look up stuff.

by Kent Porter

C-INDEX+

Product:
C-INDEX+, Version 3.1
Target:
IBM PC, PC AT, PS/2, and compatibles
Requires:
Unix System V, Xenix System V.PC; any operating system that can run Lattice C, Version 3.1; Computer Innovations C86, Version 2.30; Microsoft C 3.0, Version 3.0 or 4.0 including OS/2; Consulair C, Version 4.5
Pricing:
$395
Vendor:
Trio Systems, 2210 Wilshire Blvd., Ste. 289, Santa Monica, CA 90403; 213-394-0796

If you'd like to cut down the time required to create and implement systems that need sophisticated file-handling techniques while optimizing the amount of code and storage space needed for those systems, you should consider using C-INDEX+ from Trio Systems. *C-INDEX+ is a* full B-TREE data file management library of individually written functions. Designed specifically for C programmers, these functions can be used in any application requiring fast file creation, update, access, and maintenance schemes.

C-INDEX+ handles variable length single- or multiple-key file access of any storage file of fixed or variable length records. It also stores the file index information inside the data file itself. This index storage feature cuts down on the number of data files that are open in an application by eliminating index files. File records utilizing the C-INDEX+ functions can be retrieved sequentially, randomly, or by record number. Aside from the physical storage limits of your particular computer system, the only limitations in Version 3.1 are that single records can't be more than 10K and files can't be larger than 32 Mbyte. There are no limits on the number or format type of records in any one file, the num-

ber of fields, or the number of files that can be open at any one time.

One of the other interesting features of this system is that no reorganization of files for the removal of deleted records is necessary. All data space is automatically reclaimed by the system as long as you use variable length records. Error handling is very easy to build into your programs because each function supplied in C-INDEX+ handles the error value for you. You merely decide what you want to do when an error is found.

Both single-key and multi-key functions are supplied with the system. The single-key functions include: single and multiuser file creation, file open and file close, file buffer control, add records, change records, delete records, search and retrieve records, multiuser semaphore functions, and multiuser entry locking functions.

The multi-key functions include: single and multiuser file creation, file open and file close, add records, update records, delete records, search and retrieve records, and record locking functions. In the event of any unusual file problems, such as unexpected power outages or disk problems during file reads or writes, Trio Systems has also included a rebuild utility and an index integrity check utility to aid the programmer in the reconstruction of a suspected corrupted file. In this release, local-area networks are also supported with full byte record locking and semaphore (lock flags) functions.

I tested this product in three areas, first to compare functions that I have written and used in my own programs, second for its portability to other compilers, and third to find out what I could do with the least amount of program code and the most amount of C-INDEX+ code. In the first area of the comparison, it was immediately apparent why Trio has one product and has been working to constantly improve it. The efficiency of the C-INDEX+ code cut my program size down substantially from what it had been when I used my own functions. Converting my existing code to include the C-INDEX+ functions was time consuming because all the function calls had to be restructured or changed. When I wrote a program from scratch, however, the time differential for coding was minimal as I became more familiar with the function parameters required. In addition, when the files were recreated for the newly coded program, the storage space savings was about 25 percent due mainly to the lack of index files. Handling multikey indices and the related record add, change, and delete functions was a far easier task with C-INDEX+.

I chose the Mix C and the ALCOR C systems to test Trio's portability claim because neither of these systems were on the list of directly or indirectly supported C compilers. Since C-INDEX+ is written to very close K&R C standards, the modification time was minimal. Mix and ALCOR do not include librarian programs, which caused me to take some time making modifications, but I eventually found that this was not a major problem and successfully moved the code to both of those systems. The Trio Systems' claim that their code is portable to other systems is a valid one.

In the third area of my evaluation, the creation of the smallest amount of code, I wrote a small database system with minimal screens and data entry needs in just under 150 lines of code. The approximate breakdown of that code was as follows: 30 lines for variable definition, 30 lines for screens, 40 lines for error handling, 30 lines for C-INDEX+, and 20 lines for miscellaneous. The database stored social security numbers and names and was at best a very small skeleton of a program. I did not need to write any file handling routines, which showed me how many functions are supplied in C-INDEX+. Everything I needed was included in the system.

Although this product is extensively documented, it includes tutorial programs, and requires only calls to the various functions supplied, the user should be familiar with the C language record and pointer structures. In other words, I do not consider this to be a product for the casual C programmer but rather a product that contains a complex and very useful set of library functions for the experienced C programmer.

I heartily recommend this product to programmers who would like to cut down on development time in the data management area of their programs. Record and file handling have always been a problem in the programming arena and C-INDEX+ makes it almost effortless. With the inclusion of source code, the experienced C programmer can usually handle any adaptations to the system that might be necessary in a specific application.

by Neil Freeman

T-DebugPLUS 4.0

Product:
T-DebugPLUS 4.0
Target:
IBM PC, XT, AT, PS/2 and compatibles
Requires:
DOS 2.0 or later; 256K of free memory; Turbo Pascal 4.0; hard disk, color monitor. Extended/expanded memory recommended
Pricing:
$45; with source for $90
Vendor:
Turbo Power Software, 3109 Scotts Valley Dr., Ste. 122, Scotts Valley, CA 95066; 408-438-8608

Like many programmers, I suspect, I tend to prefer PRINT statements for running down bugs. And up until now, there hasn't been a choice with Turbo Pascal 4.0. I'd been striving for days to trap one of those ugly intermittent bugs, just about ready to give up on it, when the new T-Debug 4.0 arrived for review. Five minutes after doing the tutorial, I'd found my bug. It made a believer out of me.

The new T-Debug (despite the official name, this is how the manual refers to it) is a dressed-up version of the earlier Turbo 3.0 debugger. It does for Turbo 4.0 what CodeView does for the Microsoft languages, just as well and just as quickly.

Installation is a painless process that consists of copying a half-dozen files from the delivery diskette, running a utility that patches the two Turbo compilers and TPMAP so that they'll support mapping of local variables, and running a setup program for T-Debug. The whole thing takes about a minute.

To prepare programs for the debugger, you compile with mapping turned on (the /$T+ switch for TPC, or an Options menu selection in the environment). This tells the compiler to produce a map (.TPM) file, which T-Debug uses to find identifiers, symbols, entry points, and so forth. Compiling with the map option doesn't affect the size of the .EXE file.

Unlike the earlier T-Debug for Turbo 3.0, the new debugger is a standalone program. So if you use the Turbo environment, you'll have to leave it and run T-Debug separately. T-Debug needs about 250K of memory in addition to the memory

```
352 Var    eqptFlags : word absolute $0040:$0010;
353        videoMode : word absolute $0040:$0049;
354        egaByte   : word absolute $0040:$0087;
355        flag      : word;
356
357 Begin
358   display := ptr ($B800, 0);
359   color := false;
360   flag := (eqptFlags shr 4) and 3;
361   if (flag = 3) or (flag = 2) then
362     if videoMode = 3 then
363       if egaByte = 0 then begin          { unless proven otherwise }
 C:\TP\OPMSUNIT.PAS                                          TDEBUG 4.00
1 $48DB:$0114 choice                      $0000  00000000 00000000  0
2 $48DB:$034A color                       False
 Watch
Break at : OPMS.PAS\14  $45F3:$0011
* t 2
Break at : OPMSUNIT.PAS\358  $462C:$0D4D
* u color
* t 3
Break at : OPMSUNIT.PAS\361  $462C:$0D72
* e flag
$4947:$1FF8  $0002  00000000 00000010  2
```

Figure 2:*T-Debug divides the display into two basic windows for source and commands.*

turn. The E command examines a variable or constant. Say you have a variable called COLOR; you can type: E COLOR and T-Debug reports its current value in hex, binary, and decimal, as well as its memory address. The E command is type-sensitive, so Boolean values appear as TRUE or FALSE and characters show up as such.

The command set is quite rich, including such things as the ability to map the heap, determine memory usage, examine the stack, and decompose Pascal source into assembly language. There are also half a dozen commands for defining and managing macros, a powerful feature of the debugger.

Another powerful command set involves watchpoints. If you tell TDEBUG to watch a variable, it opens a window in the center of the screen. This is just below the register's window in the screen shot. You can watch up to eight variables (12 with EGA and VGA displays). Additionally, you can set conditional breakpoints which automatically halt the program you're debugging.

If you're debugging a large application, either extended memory or EMS may be necessary. T-Debug uses whichever is present. If you don't have one or the other, the debugger still works, but it could run out of memory.

Another recommended option is a color monitor. T-Debug makes effective use of colors, as the screen snapshot shows: register values are in red, the next line to execute is highlighted by a blue bar, commands are yellow, and so on.

T-Debug is a command-driven debugger. The lower third of the screen is reserved for dialog, and it scrolls to give a transcript of the last half-dozen interactions. Typical of debuggers, it employs a terse command structure. For example, B sets a breakpoint and -B releases it, G runs the program to the next breakpoint, T traces, and so forth. Many take modifiers: T 5 traces the next five lines of source code, as an example, and G RTN executes the current subroutine up to the point of re-

gram if a variable changes or reaches a specified value.

Some of the commands are mapped to function keys, which removes the tiresome need to type a command and press Enter each time you want to execute it. For example, F7 is the same as T (single-step). The F3 key recalls commands from a LIFO stack, displaying them so that you can edit them if necessary and re-execute by pressing Enter.

The only real complaint I have about T-Debug is that it lacks a concise reference to its many commands and function key assignments. The manual isn't terribly large—81 pages including the index—but it's a nuisance to thumb through looking for a specific command. There is on-line help, but you still have to scroll to find what you want. I finally made my own cheat sheet. The vendor should have done it for me.

T-Debug will work with dual monitors, which is an important consideration if you do a lot of graphics programming. If you only have one display, T-Debug maintains two screens: its own, and the output of the program. You toggle between them with F10. I had no difficulty tracing a graphics program on the EGA and switching back and forth between text and the drawing. However, there are some gotchas, and the manual devotes nearly four pages to a lucid discussion of them.

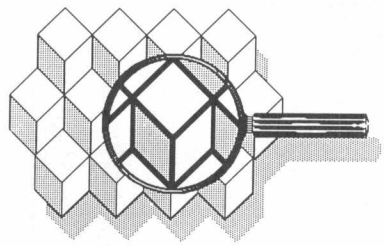
On that subject, I encountered a bug when running the EGA in 43-line text mode. Every time I switched from the T-Debug screen to the program's display, the monitor went into 25-line mode and displayed the T-Debug command area in the upper left quadrant. It didn't do any harm, but it was distracting.

T-Debug works fast and well, and it has a well-rounded set of features. If you write software in Turbo Pascal 4.0, you need this debugger.

by Kent Porter

DDJ

OF INTEREST



Language Specific Products

Applied Logic Systems has announced the release of Professional Version 1.2 of the ALS Prolog Compiler for MS-DOS computers. This version is an interactive Prolog compiler designed for programmers building complex intelligent applications and expert systems. Based upon Edinburgh-style syntax, the ALS compiler produces code which executes native reverse at 3,400 LIPS on an IBM PC XT and 31,000 LIPS on a 16 MHz Compaq 386. Features include: a module system, tail recursion optimization, garbage collection, and the ability to create stand-alone .EXE applications. A virtual code space allows programs larger than available memory to run. Price for the compiler is $499, which includes one year of free updates. Reader Service No. 20.

Applied Logic Systems Inc.
P.O. Box 90
University Station
Syracuse, NY 13210
315-471-3900

Release 5.0 of FORTRIX-C is now available from **Rapitech Systems**. FORTRIX-C is a software converter that provides automated Fortran to C. The new release adds the ability to support all of MIL-STD-1753 Fortran, all but five VMS Fortran enhancements, and virtually all ANSI-Fortran-66. Other new features include: an improved error handler with more complete diagnostics; a 30 percent reduction in the size of the output module; a post-processor that the user may elect to use to eliminate awkward (though correct) C constructs in the translated code; a makefile generator that creates a dependency script for the Unix system *make* utility; and a simple common handler designed to generate more efficient C source code under certain conditions. Reader Service No. 21.

Rapitech Systems
Montebello Corporate Park
Suffern, NY 10901
914-368-3000

BBx, from **BASIS**, is a derivative of Basic enhanced for business data processing. It offers file structures from flat files to multi-keyed files. Intrinsic locking at the record and file level prevent corruption of data during concurrent access. Designed for interactive processing, BBx provides data verification and error handling allowing development of user-proof applications. Decimal arithmetic with programmer specified precision eliminates undesired rounding of calculations. Device-independent I/O facilitates the use of terminal screen manipulation (including windowing), printer forms control, and graphics devices. All implementations of BBx are binary compatible with each other.

BBx is available for MS-DOS, PC-DOS, Xenix, and Unix on over 65 different computers. All versions of BBx include a full set of developer utilities. Reader Service No. 22.

BASIS Inc.
P.O. Box 20400
Albuquerque, NM 87154
505-821-4407

HiSoft has just released two new language products for the Atari ST: PowerBasic and FTL Modula-2.

PowerBasic is a Basic compiler that includes features such as procedures, functions, local variables, the CASE statement, WHILE and UNTIL loops, and character and integer constants. It also has short and long integer variable types, single- and double-precision floating point numbers and string variables. PowerBasic is a no-limit compiler—no program size limit, no variable size limit, no string size limit, and no array size limit.

PowerBasic will run on any ST with a disk drive and sells for £39.95.

FTL Modula-2 is a fully standard Modula-2 compiler with linker, 68000 assembler, the complete source code of most of the standard modules, and a host of utilities including a library manager and a complete CLI. Porting source code from other versions is straightforward. Some of the features specific to the ST are a menu creator and a desk accessory builder. The compiler sells for £69.95. Reader Service No. 23.

HiSoft
The Old School
Greenfield, Bedford
United Kingdom
0525-718181

Laboratory Microsystems has announced a new version of the LMI Forth Metacompiler (cross compiler) targeted to Texas Instruments' TMS34010 graphics processor. The LMI Metacompiler runs on an IBM PC, IBM PC AT, IBM PS/2, or compatible with at least 320K and MS-DOS or PC-DOS 2.0 or later. A hard disk is recommended.

The LMI Forth Metacompiler is a professional application development tool. It compiles Forth source code into a stand-alone ROMable or disk-based application. Other features of the compiler include: multi-pass, table-driven compilation; error handling; creation of ROMable or disk based applications; support of local labels and conditional compilation directives; ability to define and invocate new defining words and immediate words in the target code; optional generation of headerless code to conserve memory in the target system; optional compilation from intermediate states; built-in TMS34010 cross-assembler, using standard TI mnemonics; compatibility with the Forth-83 Standard; a detailed 200 page manual; and no royalty or resale licensing fee for well-behaved target applications. The price of the LMI Forth Metacompiler is $1,000. Reader Service No. 24.

Laboratory Microsystems Inc.
3007 Washington Blvd., Ste. 230

P.O. Box 10430
Marina del Rey, CA 90295
213-306-7412

A new version of **MIPS Software Development's** Dyalog APL is now available for the Intel 80386 chip. The version is called Dyalog APL/386 and runs under the Xenix system on the IBM PS/2, the Compaq 386, and other 80386-based computers. Dyalog APL is a second generation APL interpreter designed to run under Unix. The interpreter is written in C and offers such capabilities as: nested arrays, a full-screen editor, session manager, an interface to programs written in other languages, and auxiliary processors. The product is priced at $1,495. Reader Service No. 25.
MIPS Software Development Inc.
33493 West 14 Mile Rd., Ste. 10
Farmington Hills, MI 48331
313-661-5000

Development Tools

OASYS has introduced a family of native and cross development tools for the Intel 80386. The OASYS 80386 development tool kit runs on VAX (VMS or Ultrix), Sun, and other 68000-based systems; and 80386-based workstations such as the IBM PC and the Compaq Deskpro 386.

OASYS' 80386 compilers use global optimization techniques and employ register allocation in generation dense code. Full support is provided for two floating point units, the Intel 80387 and the Weitek 1167. Additionally, each compiler has interlanguage calling capability, for instance, C programs may call Fortran and Pascal subroutines.

The OASYS C 80386 compiler may be used with host system debuggers, such as dbx, or with the OASYS C and Fortran Source Level Debuggers for debugging in cross mode. OASYS' Designer C++, an optional C++ front-end preprocessor, is also available with the compiler. The 80386 compilers operate with other components of the tool kit: editors, profilers, and math libraries.

The OASYS Avalon 80386 Assembler/Linker and the OASYS Phar Lap 80386 Assembler and LinkLoc (linker/locator) complete the tool kit. Reader Service No. 26.
OASYS
230 Second Ave.
Waltham, MA 02154
617-890-7889

Borland International is now shipping its Turbo C Run-time Library Source, which offers complete source code to the Turbo C 1.5 library routines, with the exception of the Borland Graphics Interface and math-coprocessor emulation. Run-time Library Source is available to current Turbo C 1.5 owners for $150. Turbo C runs on the IBM PS/2 and IBM and Compaq families of personal computers and all 100 percent compatibles with 384K, PC-DOS or MS-DOS 2.0 or later, and one floppy drive. Reader Service No. 27.
Borland International
4585 Scotts Valley Dr.
Scotts Valley, CA 95066
408-438-8400

A documentation upgrade is now

available for BEYOND.BAT, **VM Personal Computing**'s package for PC software developers. BEYOND BAT provides PC developers with a script language, panel manager, and full-screen editor so developers can extend the capabilities of the DOS batch language. With the tools included in BEYOND.BAT, developers can front end existing applications, create their own applications, and pre-schedule processing so programs run unattended.

The revisions to the manual enhance its readability and provide a clearer description of the product's three running modes. Illustrations have been added throughout the manual and the index has been extended. BEYOND.BAT retails for $99. Reader Service No. 28.
VM Personal Computing
41 Kenosia Ave.
Danbury, CT 06810
800-222-VMPC

Miscellaneous Software

IGC has announced that it is shipping version 1.10 of VM/386, The Professional MultiTasker. The new version supports the IBM PS/2 and has improved hard disk support, support for VGA, and a shared RAM disk.

VM/386 is a control program that brings true multitasking to users of 80386-based personal computers by combining proven mainframe technology with the advanced architecture of the 386 chip. VM/386 uses the virtual 8086 mode of the 386 to create a series of "guest" virtual machines. VM/386 provides a separate copy of DOS, AUTOEXEC.BAT and CONFIG.SYs for each VM. The VMs run concurrently, and each one runs as if it has all of the resources of the real computer.

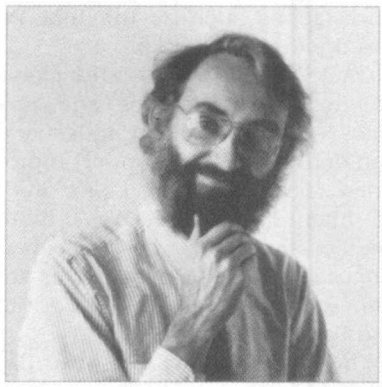
VM/386 runs on a wide variety of machines and sells for $245. Reader Service No. 29.
IGC
4800 Great America Pkwy.
Santa Clara, CA 95054-1221
408-986-8373

Tree86 Version 1.1 is now available from the **Aldridge Company**. Tree86 is a DOS extender and file management utility including unique features such as finding duplicate files, moving subdirectories, built-in mouse support, and EGA-VGA 43-50 row mode. Some enhancements include: the Xcopy command, which gives the user the ability to duplicate directory structures and copy files from one drive to another; the ability to search and work globally on multiple file specifications; the ability to use your own browse/view utility to view files; reporting of total and available expanded memory; and enhanced mouse support.

Tree86 operates on IBM PS/2, IBM PC, IBM PC XT, IBM PC AT, and compatibles with 140K and DOS 2.0 or higher. Suggested retail price is $49.95. Reader Service No. 30.
The Aldridge Company
2500 CityWest Blvd., Ste. 575
Houston, TX 77042
713-953-1940

DDJ

SWAINE'S FLAMES



Underlying the hype over HyperCard and CD-ROM there are some pivotal truths, one of these being that well-packaged information is becoming a bigger commodity than ever before. Those software developers and information brokers who can pick up on each others' skills and resources will be able to create a new kind of product, a blend of information and code that goes well beyond what we have seen to date in stackware (better call it heapware).

How these people will create this marvelous new kind of product I do not say because I do not know. However, as an information broker, I must pass along to you software developers this valuable insight into the laws on libel.

The court that heard the appeal in the *Jerry Falwell* vs. *Hustler* magazine case ruled that *Hustler* magazine had not libeled Falwell because *Hustler's* imputation of sexual impropriety to an evangelist was implausible.

It's the Implausibility Defense, and I see a great future for it. Just make sure your claims are implausible.

But this is all just common sense. You don't need my information broker's insights.

Many truly valuable insights for software developers were being dispensed at Miller Freeman's Software Development '88 this year.

Keynote speaker Jon Bentley outlined his three principles of programming: prototyping, profiling, and the use of little languages.

We heard more about prototyping from Dan Bricklin, who gave some reasons for prototyping—that is, to get nonprogramming experts and potential users involved in the design, to get good products done faster and to weed out bad ideas quickly, and to impress funding sources. Then, realizing he was preaching to the converted, he gave one typology of prototyping methods (algorithm prototyping, functionality prototyping, appearance prototyping) and some techniques for each type.

Several speakers addressed profiling. Chuck Duff, for one, talked about using the profiling capabilities of Actor to identify the areas of the code that are hogging the processor, so you can selectively change dynamic bindings to static for efficiency.

And William Barrett expanded on the theme of little languages, mentioning that developers working in the Macintosh or VAX VMS or PC AT environment who want the combined functions of Unix *lex* and *yacc* should write to QCAD Systems in San Jose and ask about Qparser+.

And design methodologies. Miller Freeman is big on them. Larry Constantine declared the two powerful principles at the heart of every system design approach to be: Take notes! Draw pictures! Himself a design methodologist, Constantine admitted that modeling a system that doesn't yet exist is a creative endeavor that doesn't yield to techniques that are too structured.

Edward Yourdon was there with copies of his new newsletter, *American Programmer*. It was full of plausible grim prognostications about Japanese and European programmers taking the software industry away from the American programmer. Contact number: 212-769-9460.

Yourdon is no more depressing than my cousin Corbett, though, who recently told me this tale of software woe.

Immediately after playing Chris Crawford's Balance of Power, Corbett decided to write his own game program. It would allow players to choose countries and replay the major wars of the twentieth century. Players could develop nuclear weapons, but if they used them, the program would crash, formatting the hard disk. But the war gaming would be only a tactic in the real strategic goal of the game: global economic dominance.

Midway through the development process, *Esquire* magazine pronounced the death of the Yuppie. It didn't stop Corbett, though. He's hoping that the culture of greed will hang in there long enough for him to make a buck off this product.

It might, but he has now encountered a most frustrating bug. He had gotten as far as developing a testable version of the product and had written an autoplay program to simulate various player strategies when The Bug appeared.

It seems that every time the game is played, the result is the same. The countries that lose the last major war before the atomic age are subsequently prohibited by the victors from developing nuclear arsenals. While the victors invest in nuclear weapons that they will not be able to use without destroying the system, the losers concentrate on education and usable technologies, and achieve global economic dominance in the next round. Same result every time. Boring.

Corbett sees the trap quite clearly. He just doesn't see any way out.

Michael Swaine

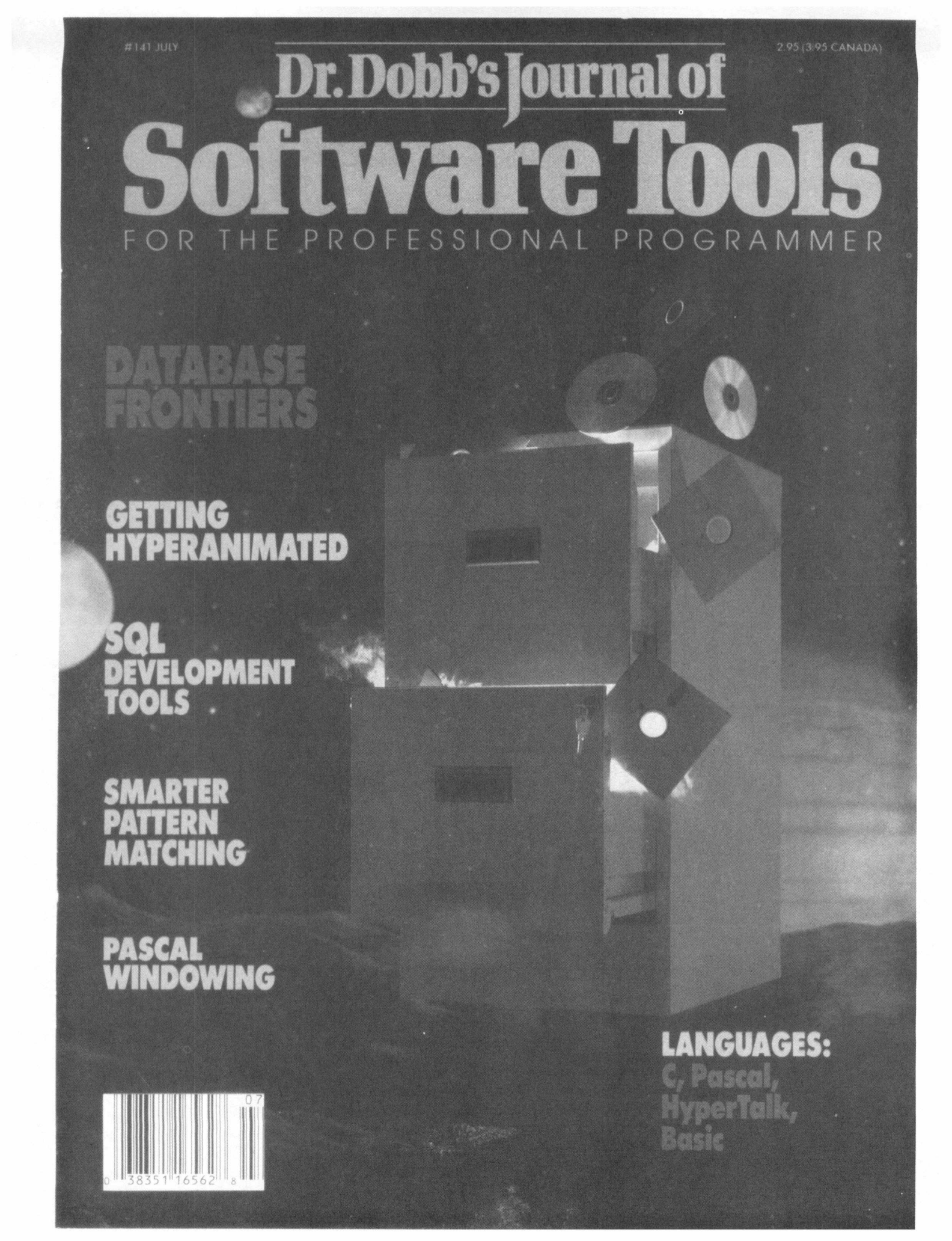
Michael Swaine
editor-at-large

#141 JULY

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER



DATABASE FRONTIERS

GETTING HYPERANIMATED

SQL DEVELOPMENT TOOLS

SMARTER PATTERN MATCHING

PASCAL WINDOWING

LANGUAGES:
C, Pascal, HyperTalk, Basic

0 38351 16562 8 07

ARTICLES

Getting Hyper ▶ **HyperAnimation** **18**
by Elon Gasper
If you thought talking heads like Max Headroom were science fiction, guess again. In this article Elon talks about the technology behind HyperAnimator, a Macintosh product that lets you add synchronized animated heads and sound to produce a lifelike effect.

SQL ▶ **SQL Development Tools** **36**
by Mike Geary
Mike talks about SQL and provides an introduction to SQLWindows, an SQL development tool he wrote. Along the way he also talks about developing for the Microsoft Windows and the Macintosh environments.

Matching Patterns ▶ **Pattern Matching by Gestalt** **46**
by John W. Ratcliff
String comparison routines are often limited to finding exact matches. John describes an algorithm (implemented in assembly language) that gives matches as percentages.

REVIEW

EXAMINING ROOM **52**
by Bruce Tonkin
Heated competition has led to increasingly more powerful Basic compilers. Bruce reports on "the state of Basic" with an examination of Microsoft's Basic 6.0 and Quick Basic's 3.0 and 4.0, as well as Borland's Turbo Basic 1.1.

COLUMNS

C CHEST **116**
by Allen Holub
This month, Allen takes a look at shell redirection and offers two programs for dealing with shell archives in the Unix system and MS-DOS. He also has some thoughts on the latest book from Kernighan and Ritchie.

Pascal Pop-ups ▶ **STRUCTURED PROGRAMMING** **122**
by Kent Porter
Back in our November 1987 issue, Kent demonstrated how to code pop-ups, pull downs, and menu bars in Turbo C. Now he's back with an improved version for Turbo Pascal.

PROGRAMMING PARADIGMS **130**
by Michael Swaine
Paradigms in Collision. This month, Mike considers the parallels (so to speak) between the paradigm shifts of cognitive psychology in the 1970s and the growing interest in neural networks.

FORUM

EDITORIAL **6**
by Jonathan Erickson

RUNNING LIGHT **8**
by Tyler Sperry

LETTERS **12**
by you

CARTOON **14**
by Joe Sikoryak

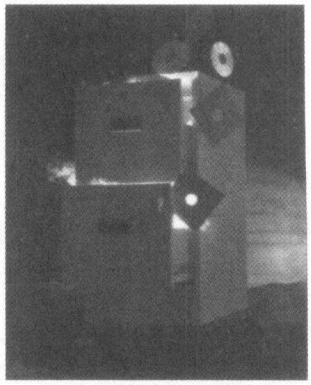
SWAINE'S FLAMES **144**
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST **138**
brief product descriptions

ADVERTISER INDEX **129**
Where to go for more information on products.

PROGRAMMER'S MARKET PLACE **140**
classified ads



About the Cover

Ever had one of those days where everything seems to have disappeared on you? Do your friends complain about your spacing out? Well then maybe you're ready for the next programming frontier: database technology.

Next Issue

Our annual C issue includes a comprehensive review of optimizing C compilers by Richard Relph, as well as some utility programs for managing C projects. And then there's the usual good stuff.

EDITORIAL

There's no question that OS/2 has a promising future. The significant word here, however, is *future* and IBM's OS/2 exhibit at Spring Comdex underscored this fact. Yes, IBM was passing out a book that listed hundreds of *announced* OS/2 applications from third-party developers. And yes, the OS/2 booth was one of the stars of the show. But of the 100 OS/2 applications that were supposed to be on exhibit (of which only about 50 or so ended up being demonstrated), less than 30 were shipping, and it seems that it will be at least another year or two before the promise of OS/2 is closer to being fulfilled.

Software developers were frank in their explanation of why there is a relative dearth of released products for OS/2. The main reason, say developers, is simply a lack of demand by PC users.

But before user demand for OS2 applications can be stimulated, several pieces of a puzzle must fall into place. For one thing, systems that run the applications most people want require an 80386 CPU, and Intel, which is currently the only source for 80386s, has not been able to produce enough chips at an affordable price to meet current demands. Nor has the company made any moves to second source the 386; instead it is fighting in court with companies such as AMD that want second-source rights. To its credit, Intel has recently opened up several new plants dedicated to manufacturing 80386s. Unfortunately, some of these plants will probably be producing the "P9," a 386 that supposedly will have a 16-bit data bus. (Intel take note: Even GM realized no one wanted a 6-cylinder Corvette.) Because of Intel's strategy, an 80386 costs around $300, about ten times the cost of an 80286.

Second, the amount of memory required to run OS/2 applications seems staggering, at least to those programmers who were weaned on CP/M. If you want just to boot OS/2, you'll need 1.5 Mbytes of RAM; if you want to run an application, you'll need at least 3 Mbytes. Memory, particularly in large quantities, is expensive right now (1-Mbyte SIMM chips are selling for about $500, double what they cost six months ago), and it's unlikely that it will become cheaper over the coming months.

Finally, Microsoft will have to release a 386 version of OS/2. What the current implementation of OS/2 (written for the 80286) provides that DOS does not is multitasking and the ability to handle a large-memory model. What OS/2-386 will provide is support for a virtual 8086 mode (so that you can have multiple 8086 applications running at once, a la Windows/386), demand paging, and 32-bit memory addressing with large data objects. This, along with a graphical interface like Presentation Manager's, is what I want, and I'll bet that's what most other users want too. When this is available and when 386-based PCs are affordable, end-users will clamor for OS/2 and its applications. Microsoft, however, says it won't start releasing OS/2-386 software development kits until mid-1989.

What this means is that DOS will continue to be a dominant force while other operating systems—Unix and the Macintosh OS, for instance—will continue to attract more and more followers. Unix advocates in particular are making major advances, with even Microsoft admitting that demand for Xenix soared after the details of OS/2 were finally released last year. In the meantime, the door is still open for alternate operating systems, and it will remain open longer than many of us originally expected.

Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editors *Allen Holub, Richard Relph, Kent Porter*
Contributing Editors *Stan Krute, Martin Tracy*
Copy Editors *Lorraine Aochi, Carol Dondrea, Kevin Shafer, Rhoda Simmons*
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Lynn Sanford*
Typesetter *Mary E. Lopez*
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Asst. Circulation Manager *Andrea Weingart*
Newsstand Manager *Sarah Frisbie*
Direct Marketing *Kathleen Shay*
Fulfillment Coordinator *Francesca Martin*

Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, with two special issues per year by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA 800-321-3333; in CA 619-485-9623 or 566-6947. For book/software orders call 415-366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

In keeping with this issue's theme of exploring database technology, I thought I'd share a secret with you. It is not a particularly big or terrible secret but rather a confession of prejudice—a prejudice I share with many of you who read this magazine.

I hate databases.

It is an irrational reaction, I must admit. Like most *DDJ* readers, I'm happiest when I'm programming until the early morning hours. But for some reason, if you mention databases my love of programming is replaced with a sense of dread. My palms start sweating and my vision blurs. Nightmares appear of my personal Hell: eternity spent working on IBM's master payroll package with nothing but dBase II and a floppy-based CP/M machine. Shudder.

So, you ask, if database programming is terminally boring, why devote an issue to the topic? The answer is that it is the future of database technology that is exciting. Back in the early days of micros, the challenge was making our machines do useful work with only a cassette tape drive and 8K of RAM. These days we're facing an entirely different challenge: making the most of machines with optical drives and megabytes of RAM. One of these challenges is hinted at in this month's lead article. Elon Gasper and the crew at Bright Star Technology are convinced that animation is the next big wave for computers. I'm naturally skeptical, and my first thought was that talking heads reached their peak years ago with Steve Hall's Talking Moose for the Macintosh. Still, I've been wrong before. Anyone who's seen a demonstration of the DVI technology for CD-ROM, for example, must realize that we've only begun to explore the potential of mixing video and computers.

Moving on to the practical aspects of managing information, one of my problems with most database products is that they always seem to be more trouble than they're worth. A flat, text database is all I need, thank you. I don't want to program in a database language, and I don't want to spend hours wading through forms and menus. If you're of a similar persuasion, you might want to check out a product called Memory Mate. Originally a shareware product written by Michael Fremont, the program has recently been improved and remarketed by Broderbund. I've been using the product for a couple of years now and couldn't function without it. It doesn't overwhelm you with bells and whistles, but it does have two important features: it's simple and it's fast.

As a final note, I'd like to mention a new arrival in the category of multitasking windowing pageware, otherwise known as books. This last month I received a book that immediately went on the shelf within arm's reach—right next to my well-worn copies of Ray Duncan's *Advanced MS-DOS* and Peter Norton's *Programmer's Guide to the IBM PC*. The new book, by Thom Hogan, is called *The Programmer's PC Sourcebook* (Microsoft Press), and it lives up to the name. In 525 pages Hogan gives you all the detailed information and tables that you previously had scattered in a dozen volumes. We're talking hard data here; the pages are filled with tables listing everything from BIOS and DOS interrupts to card connector pinouts, from data file formats to physical addresses for the PC's support chips. Best of all, Hogan does the unthinkable: he cites his sources. Highly recommended.

Tyler Sperry

Tyler Sperry
editor

ARCHIVES

Ten Years ago in *DDJ*

"Computers have been consumer products since 1975. Individuals involved in the consumer computer industry estimate that there are 50,000–20,000 general-purpose computers installed in peoples' homes. Thus far, these have been exciting, educational, challenging toys—to the extent that they have been used for non-commercial, non-tax-deductible purposes. Their primary use as a consumer product has been in non-arithmetic or minimal-computation applications. One difference between their being an exciting toy, and making them into an obviously useful consumer product, of value to the general public, is the ability to attach them to large data-bases of interest and use to the general public."—*Jim C. Warren, Jr., "The Digicast Project: An Immediately Viable Broadcast-based Information Utility For the General Public,"* DDJ, *October 1978.*

We Want . . . Information

"Computers and their owners are at the leading edge of this information era. Whether they will remain there long enough for us humans to more fully realize our potential for self-determination depends on several things.

First, the computer user must know what information he requires. Then he needs to learn who has it, how to access it, and finally, what to do with it. . . . —*"Marlin Ouverson, "Editorial,"* DDJ, *May 1981.*

Through a Glass Darkly

"Windows are used to split a computer screen into several possible overlapping viewing areas. Like a tiny screen, each window may be used to show a separate program or task.

Windows are used extensively in the Xerox Star, VisiCorp's new VisiON operating system for the IBM PC, and are the basis of the Apple Lisa and the soon to be announced MacIntosh computers."—*Edward Mitchell, "A Simple Window Package,"* DDJ, *January 1984.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Parallel Programming: An Old Hat

Dear *DDJ*,

For goodness sake, in Michael Swaine's new column (Programming Paradigms, May 1988) please do not reinvent and rediscover parallel programming. Pease read standard books (including mine!) on the subject, e.g.: *Concurrent Euclid, the Unix System and Tunis* (Holt, Addison-Wesley), and *Operating System Concepts* (Peterson & Silberschatz, Addison-Wesley).

At the University of Toronto, parallel programming has been taught using a high level concurrent language since 1974. We are now using the Turing Plus language for this purpose.

Professor R.C. Holt
Toronto, Canada

Porting Virtual Arrays

Dear *DDJ*,

Although "Virtual Arrays in C" by Mark Tichenor (May, 1988) may run on a PC when compiled with Turbo C, it will definitely not run on many other machines. This is because Tichenor has hard-coded the lengths of integers, long integers, and so on into the majority of his software. For example, the statement:

```
fwrite( &size, 4, 1, f );
```

should be:

```
fwrite( &size, sizeof size, 1, f);
```

or maybe better yet:

```
(void) fwrite( &size, sizeof size, 1, f );
```

The problem is that in this statement, the size of the variable is written out explicitly. If the software is ported to some other hardware, integers and longs may be a different size. On a VAX, for example, both are 4 bytes; this would cause the statement "fwrite(&rec_size, 2, . . ." to no longer be correct.

In Listing Two the following software is shown:

```
for( i = 0; i < v_array ->
                        buf_size; i++ )
    *buf_ptr++ = filchar;
```

I am at a loss as to why this is not:

```
memset( buf_ptr, filchar, (size_t)
                v_array -> buf_size );
```

instead. Although this may seem like a small matter, the run-time library copy of *memset( )* would likely be faster than executing the loop above. The reason is that quite a few processors have block-move or similar instructions which can be utilized inside *memset( )* to rapidly set the area. Another possibility is that *memset( )* could be treated as an in-line function, with the compiler generating the block-move without a function call at all.

Bill Mahoney
Omaha, Neb.

Writing a Loader for EXE Files

Dear *DDJ*,

We subscribe to your magazine and we often get useful suggestions from the articles. However, we now have a problem that has not been addressed, to our knowledge, in *DDJ*. Perhaps you can help us.

We need to write a loader for EXE programs that performs the same loading function as the DOS loader. A companion program will begin execution of the program (after massaging the data for our own purposes). We have tried to use the DOS function 75 to do the loading, but we do not have adequate information to complete our task and begin program execution. If we had the exact, detailed data structure of EXE files, we could write our own loader, but we still need information on starting the program. Our books only give cursory, general information. Do you know how we might obtain such information?

Also, as part of the same project we are writing a DOS device driver but having trouble debugging. Our driver works fine as a separate EXE program, but when running as a device driver it fails. DOS DEBUG is useless for device drivers, so we need a third-party debugger.

Dr. Gary D. Hornbuckle
Hornbuckle Engineering
Pacific Grove, Calif.

Porting Small C to the Mac

Dear *DDJ*,

I have undertaken porting the Small C compiler, as well as Small Mac and Small Link for CP/M-80 to the Macintosh. I took to heart Jim Hendrix' comment that the system could be used easily to create a cross compiler, and I'm trying to create a complete cross-development system. I would hope to be able to generate fully executable CP/M modules, ready for downloading.

My first question is this: Has such a thing been done already? Are there any readers who have experience porting Small C to other environments for purposes of cross-development? Can you offer me any tips?

My second question: At one time there were CP/M emulators for the Macintosh. Am I right in assuming that there are no CP/M emulators on the market now? Are there any readers who would be willing to sell me an old emulator?

Matthew Snyder
South Bend, Ind.
UUCP: inuxc!uivax!ndmath!matt
Bitnet: mjs@irishvm
(mjs@irishvm.bitnet@wiscvm.arpa)
CompuServe 71450,2606
GEnie: MSNYDER

Put the Info Where?

Dear *DDJ*,

The discussion of where to store a programs' information (C Chest, February 1988) was fun, but left a bit much to the imagination. Specifically, how does one force the options structure to the physical end of the file? I presume by creating a file which only contains the structure, then referencing it last in the link list. But then why is *stdio.h* and

HyperAnimation fills the gaps in Ed "Lonely Guy" Grimezyk's social calendar.

fcntl.h included?

Regarding the checksum mentioned, I have inspected the header block of several files and found the checksum bytes to be zeros in all except the DOS utility files. Furthermore, changing a byte in a file had no effect on the ability of DOS to load and run it. I suspect that this aspect of the EXE header must be a holdover from older times.

Thomas Gettys
Lafayette, Col.

Setting Attributes

Dear *DDJ*,

Ignoring for a second the problems associated with synchronization on the CGA, do all IBM color cards work at segment B800 for text? In other words, if I write directly to the screen, will CGA, EGA, and VGA all use the same character-attribute bytes text-page format starting at B800:0? Are there any differences in the character-attribute technique with regards to VGA?

Will Estes
Saratoga, Calif.

Kent Porter replies:

The EGA and VGA both detect the kind of monitor attached and alter their behavior accordingly. If you have a monochrome tube hooked to an EGA or VGA, the adapter pretends that it's an MDA and starts video memory at segment B000.

The monochrome attributes then apply. When a color display is connected, the same text attributes apply to CGA, EGA, and VGA. All three behave identically in text mode. Of course, EGA has a 43-line mode and VGA has both 43-line and a 50-line mode, and you can also play some games with the number of columns. Mainly, however, it's only in graphics modes that they differ.

An excellent source of information about these matters is Bob Jourdain's *Programmer's Problem Solver* (Brady/Prentice Hall). Others are Ray Duncan's *Advanced MS-DOS* (Microsoft Press) and R.A. King's *The IBM PC-DOS Handbook*. If you're doing serious programming, you need at least one, and preferably all, of these references, as well as others.

GETTING A HEAD WITH HYPERANIMATION

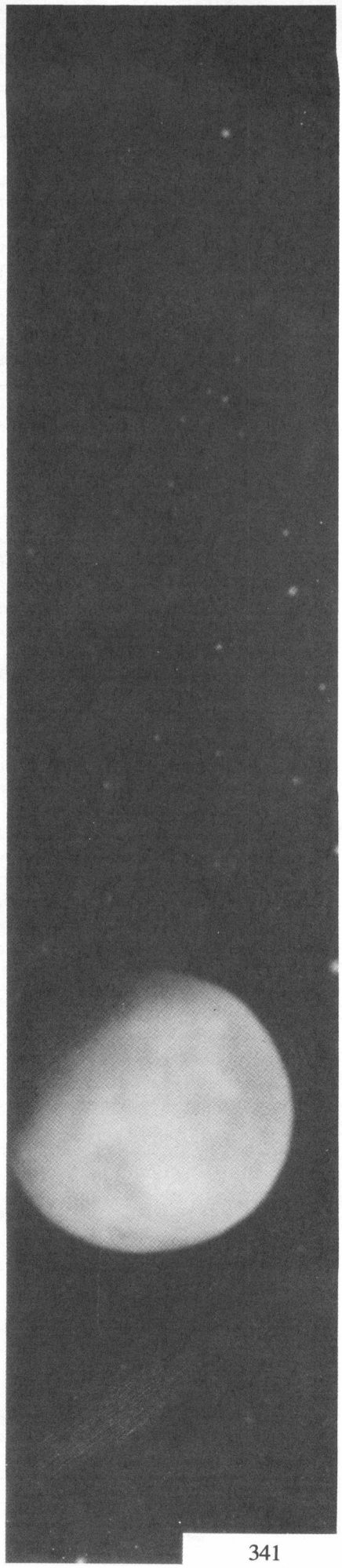
by Elon Gasper

Within the next few years, databases will do more than simply provide information in the static, sequential form we've become accustomed to. Advances in both software and hardware technologies will enable developers to provide new ways of presenting information, methods that include images and sound, as well as the familiar alphanumeric data.

One approach envisioned by futurists is a "talking head" computer interface. Apple has recently suggested that in the twenty-first century interface such as this will be an important part of its "Knowledge Navigator," HyperCard, and other powerful software applications that integrate hyper*media, simulation* and artificial intelligence.[1] (See accompanying text box.) A talking head is essentilally the life-like image of a human face which interacts with the computer user. Alternately referred to as a synthetic actor or an anthropomorphic agent, it will be one of the basic means by which future PC users interact with their computers.

In this article, I will describe a way to create synthetic actors using a technology I call HyperAnimation that combines and coordinates images, sound, symbols, and data to present information in the form of a talking head on the computer screen. However, the HyperAnimation system is more than just the front-end to a database; it also includes a database engine and a set of software tools that lets developers digitize images, bring those images onto the screen, and make the image say anything you want. What I will do in this article is first to describe the components of the HyperAnimation development system, then discuss how those components work together.

Elon Gasper is the president of Bright Star Technology Inc. He is a specialist in interactive graphics and real-time programming for small computers. Elon previously was software development manager for a UCLA audiovisual research department and has also taught in the California State University System. He may be reached at 14450 NE 29th St., Bellevue, WA 98007.



Similarities to Hypertext

Early applications of computer-augmented text editing and storage were limited to the linear-thought organizational methods of paper-based systems. Hypertext frees the user of words from the sequential nature of their embodiment of words on paper. An analogous situation has developed for animation technologies. From celluloid strips, to videotape, and on to desktop presentation software, sequential methods have prevailed. HyperAnimation is the first general-purpose system for random access and display of images on a frame-by-frame basis, a system that is organized and synchronized with sounds. Like Hypertext, HyperAnimation technology enables its users to transcend the limitations of linearity and to create a whole new realm of possibilities.

Specifically, with HyperAnimation software the computer can intelligently coordinate image display with synthetic speech so that a representation of a human being, a cartoon character, or even a robot, can deliver lines never spoken before, complete with appropriate facial movement. This ability to rigorously combine randomly accessed images and speech fragments under control of an interactive computer program can be summed up in three words: "Read my lips!"

The Applications

Each generation of human interface is more intuitive for users because of a better fit to our innate human abilities. The first generation was simply switches and lights; the second, keyboards and character output; the third employed graphics metaphors and a pointing device. Voice synthesis and recognition is the fourth. With HyperAnimation capabilities, the addition of a synthetic actor's talking face further enhances the communications bandwidth, introducing the fifth generation human inter"face".

An opportunity exists for applying HyperAnimation anywhere that humans interact with computer technology. The countless possibilities include interactive entertainment, education and training, telecommunications, robotics, and, of course, new applications software.

The remainder of this article provides an overview of HyperAnimation technology: the paradigms used in implementations and the basic structure of the control software. Bright Star's Talking Tiles educational program (which uses an anthropomorphic agent as a simulated teacher) provides an example of HyperAnimation capabilities and potential, while the HyperAnimator is used to describe how to experiment with synthetic actors by using HyperCard.

New Tools for Actors

HyperAnimation technology is based on a descriptive authoring language called RAVEL (the Real-time/random-access Animation and Vivification Engine Language) and its associated run-time system RAVE. The RAVEL language contains statements to describe any system of symbols, sounds, and facial movements that a designer can use to weave the patterns of human speech and facial images into a talking face capable of reading a script stored in ASCII text form. This enables a designer to ravel the patterns of human communications. This capability decomposes an individual's facial images and sounds into constituent parts, and specifies how these visual and audio threads are to be dynamically rewoven into the image and sound of that person (or thing) saying (or doing) something else.

Synthetic actors can read from the same cheaply stored (and easily modified) text scripts, can use different voices, and can speak different languages. Interchangeable models of celebrities, politicians, cartoon characters, or even yourself can be plugged into applications programs as easily as fonts into a PostScript document, or peripherals into an SCSI port. This interchangeability enables quick prototyping of synthetic actor designs.

In addition to RAVE and RAVEL, we have created a number of tools. These tools were written to create the most general context possible for the development and utilization of synthetic actors. These tools serve primarily as anthropomorphic agents and simulated teachers, and especially for language learning.

For example, a device-driver version of RAVE (currently in field testing) can be used by any Macintosh application. The driver has been linked to Apple's HyperCard with an XCMD to open the use of synthetic actors to any stackware creator. An early version of this product, the HyperAnimator, was demonstrated in January at the MacWorld Expo show. This article later discusses how RAVE is invoked in this context from Hypertalk to summon synthetic actors. (An example script is given in Listing One, on page 66.)

We have also released some educational software that use HyperAnimation. Alphabet Blocks, for example, teaches basic phonics using a cartoon synthetic actor—a magical talking elf who talks to and interacts with children as they play with simulated objects depicted on the screen. In Talking Tiles, a digitized teacher shows how the pronunciation of words is related phonetically to spelling of the words. This application is described in more detail because it illustrates how the capabilities of RAVE can be maximized.

RAVEing to the Max

Talking Tiles is a general-purpose learning tool intended primarily for teaching language skills. In it the animated lip-synchronized synthetic actor functions as a "simulated" teacher who instructs the student by using simulated anagram tiles. When the student selects a tile, the simulated teacher pronounces the proper sound (or sounds) associated with that letter (or set of letters). The selected tile may then be dragged onto the playing field to begin a word or added to an existing word (Figure 1, page 22). The *simulated* teacher then pronounces the resulting combination. The unlimited vocabulary enables a student to experiment by constructing sequences of letters to make words and even to make sentences.

Letters in the tiles are highlighted at appropriate times to show which letters made which sounds and why. The pronunciation of each word proceeds in

(continued from page 20)

synchrony with a wave of highlighting that moves from left to right (in the case of English). This is coordinated with the speech sounds so that each letter in the word is highlighted during the audio presentation of the part of the combined sound during which its contribution is most prominent. This process is called *orthophonetic animation*.

The "simulated" teacher can sound out words by pronouncing component sounds of the word in sequence with unblended speech. The letter (or letter combinations) responsible for the sound are simultaneously highlighted. Letters influencing the sound made by a particular letter in a word may also be indicated by underlining. The visual emphasis shows the student why a letter made its characteristic sound. The synthetic actor may even explain or comment on the letters and sounds. These functions are under the control of the RAVEL language, so the functions are fully independent of the human language (or set of symbols) being used.

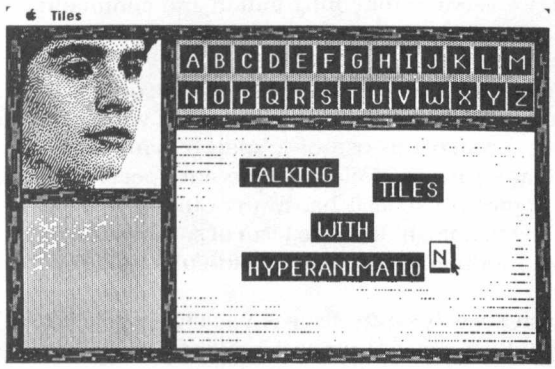


Figure 1: *Talking Tiles, the talking digitized teacher has rigorously synchronized lip movements. The teacher shows how the pronunciation of words is related phonetically to their spellings as the students manipulate simulated anagram tiles to make words.*

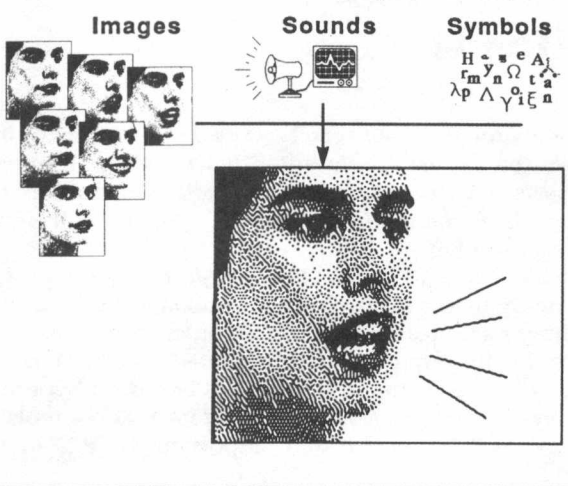


Figure 2: *HyperAnimation technology offers a general method to synchronize images, sounds, and the symbols and associated with them. Any number of images can be coordinated with speech to create synthetic actors.*

The talking head of the synthetic actor provides synchronized moving lips (as well as other head and body movements) to provide the auditory and visual clues present in human speech. The synthetic actor's functions include enhancing the recognition of its synthetic speech with synchronized lip movements and other gestures. The talking head also makes the learning game more attractive and emotionally appealing. The talking head encourages imitation by demonstrating the forming of sounds with the mouth. Furthermore, the synthetic actor serves as a master of ceremonies for the learning program by explaining and demonstrating the use of the program and interrupting periods of inactivity to wake up or encourage the user.

Many extensions of this expert system for learning have been contemplated and planned. They all rely on RAVE actors' inherent understanding of the reading/speaking universe of discourse. For purposes of this article, the system is discussed simply as an example of the sorts of revolutionary applications that RAVE enables with its anthropomorphic agents.

How the Process Works

Before RAVE software can create HyperAnimation, the RAVEL author must describe the basic components of each synthetic actor. Images, sounds, and symbols are the three most obvious elements (Figure 2, this page). RAVEL statements define and link them together in a web of relationships. Each synthetic actor definition integrating these components is called a "model."

For each model, the designer specifies the behavior patterns of the talking synthetic actor (its voice and associated images) and how they are to be coordinated. An arbitrary number of associated sounds, symbols, and drawn (or digitized) images may be involved through the use of its orthography and phonology encoding scheme designed to handle any human language. This enables the potential realism of the portrayal of the synthetic actor and uses these techniques to be essentially unlimited.

The design goal was to make HyperAnimation independent of the technologies it integrates. Differences in human language (as well as the details of the display and sound production methods) are hidden from the application. A change in any one of these characteristics is handled at the RAVEL level by the synthetic actor animation designer. Thus, HyperAnimation applications can migrate upward as constituent technologies progress.

Image

The RAVEL designer provides a database of source images from which the RAVE run-time system creates synthetic actors. Any number of images may be used. They may be drawn by an artist or digitized from live subjects, claymation-type sculptures (such as television advertising raisins), or videotape recordings. More images mean more realistic synthetic actors. Fewer images mean less storage used and faster development cycles because application prototyping can be done by using a

(continued from page 22)

subset of a larger production version model.

The minimum number of images that can crudely simulate lip-synch is the degenerate case of two: one with the mouth open, the other shut. Surprisingly, even this can suffice, if only for crude humorous presentations. Eight images corresponding to various distinctive speech articulations are sufficient to create an acceptably realistic synthetic actor. Using the HyperAnimation tools and such a model, a designer may spend but a few minutes to digitize any person, to put that representation up on the screen, and to have that representation talking.

More complex models use additional images to enhance the smoothness of the presentation, to show separate exact articulations for each phonetic component for teaching language skills (i.e., literacy, lip-reading, remediation, and therapy), to depict emotion, and for humorous enactments where the synthetic actor changes as it speaks.

The RAVEL language and its associated tools enable artists and animators to produce, manage, modify, and manipulate databases that represent the sets of images which make up models. Each database can be structured to reflect hierarchical relationships among sets of images in order to provide for submodels. For instance, one set can depict the actor in profile, another in a face-on view. Transition images can then animate the turning process itself. RAVEL statements designate all the images and tell how they are to function with sounds and with each other. An in-between operator in RAVEL can specify images that are inserted to add smoothness in transitions. Provision has also been made for automatic generation of in betweens that use a motion-blur algorithm currently under development.[2]

Special RAVEL statements tell the compiler each model's image size, pixelation, and the name of the file in which the images are stored. At run time, a RAVE routine opens each file, reads it and prepares the data structures. In current implementations this is all done in RAM; later versions will use the submodel structuring to buffer sets of images as needed. This will be especially useful with the very large models (and thousands of images) stored on optical media.

RAVE is set up to be as independent of the animation means as possible. Given the capabilities of the installed base of Macintosh computers, it is appropriate only to store and display bit-mapped screen images. Provisions have been made in RAVE and RAVEL for parametric descriptions[3] to be used instead of bitmaps so that processors with sufficient speed can create the images themselves by using mathematic models of the anatomy of the face. Another possible animation means is robotic. A lip-synced robot interfaced to the Macintosh has been demonstrated. It uses variants of RAVEL statements that describe servomotor configuration and commands.

Sounds

RAVE actors can function with prerecorded digitized sounds. But in order to have unlimited vocabulary, they speak through the use of a speech synthesizer. Many companies have developed different speech synthesizers. Some use special hardware; others (like Macintalk on the Macintosh) take the form of software-only device-driver modules resident on a particular host microcomputer.

Each speech synthesizer is usually associated with a facility to translate human language source text to the phonetic codes used to control it. That process will be discussed separately in the next section. For now, consider only the speech synthesizer proper, (i.e., the part that just pronounces the phonetic codes sent to it).

Though their internal methods vary greatly, each

Apple's Head Talks About Talking Heads

While HyperAnimation is a development tool that currently provides programmers with a talking-head front end to databases, the Knowledge Navigator is Apple Computer's view of how a similar technology might be implemented in the future. In fact, Apple chairman John Sculley has even gone so far as to characterize the talking head technology as the basis for the Macintosh of the twenty-first century.

Apple introduced the concept earlier this year in a company-produced film entitled "The Knowledge Navigator." At the opening of the film, the PC (which looks surprisingly like descriptions of Alan Kay's long talked about Dyna Book) has a screen with a Mac-like Menu Bar across the top and a "knowledge assistant" (i.e., talking head) in the corner. The user retrieves a map of South America from a database by simply telling the talking head to get it and, by pointing to various parts of the map, the user can search other databases, including those that contain relevant articles. When the user requests a specific article by a particular author, the talking head supplies the correct title as well as an abstract about it.

Sculley claims that the Knowledge Navigator project is more than just marketing hype and states that Apple's engineers are currently using computer simulation (presumably on Apple's Cray computer) to develop the project.[1] He adds that the project is based on hypermedia, simulation, and artificial intelligence: Three tools he believes will be increasingly important for PCs in the future.

"This film is not a fantasy," he said. "It is based on work taking place at corporate and university research centers today." —eds.

(continued from page 24)

speech synthesis unit appears to the calling program as two parts: a way to produce relatively short phonetic segments of speech, as well as a way to concatenate and blend them together. This thumbnail description is a drastic oversimplification of a complex situation.[4,5]

There is no standard way to break up speech into segments, nor any agreed upon set of codes to denote the segments. Indeed, different ways (words, phonemes, syllables, and so forth) work better for different human languages and design constraints (such as memory and speed). So each synthesizer uses its own segmentation methodology with idiosyncratic encoding terminology. To make matters worse, the encodings often vary in length in an attempt to be mnemonic. For example, Macintalk uses strings of approximately 60 possible codes, each of which is comprised of one or two ASCII characters.

Especially in the case of microcomputer software, each developer of a speech synthesizer has tended to regard its product as the "be-all" and "end-all" of speech synthesis. For instance, the system-resident speech synthesis software module on the Amiga is referred to as the Narrator Device. The RAVE/RAVEL system takes a broader perspective by characterizing each speech synthesizer as an instance of a narrator device. Bright Star wanted applications that use RAVE synthetic actors always to be able to take advantage of the state of the art in speech generation without recoding at the application level. Thus, a Narrator Device Integrator (NDI) was built into the RAVE system. The characteristics of a particular speech synthesizer or system of digitized sounds are then described by using special RAVEL statements that program the NDI functions. This also provides for using different speech-generation products as different voices for separate actors appearing together in an application.

The calling program never needs to know which speech synthesizer is being used. This information is hidden from it by RAVE, which assigns its own unique fixed-size encodings (called "phocodes") to speech synthesizer segments. RAVE and its calling program pass phocode strings back and forth to communicate. This enables the application to work with different synthesizers and human languages simply by specifying a different model.

Figure 3: *Talking Tiles with a Greek language tile set loaded.*

Symbols

RAVE synthetic actors are designed to be able to read and speak any human language. The nature of the symbols, sounds, and their relationships are specified in RAVEL in order to be transparent to the calling program. The design goal is to enable any organized set of symbols of visual symbols associated with sounds to be programmable synchronized with facial animation and speech (or other sounds). Letters, words, sign language, mathematics, or scientific symbols are examples.

In order to provide for human language independence, text-to-phonetic translation functionality is built into RAVE. This is accomplished by specifying a set of rules (productions) to govern each particular translation. For many human languages, these rules are provided in a form similar to those of Elovitz et al.[6] Their methods are extended by several additions (including statements that define categories of characters) to achieve generality across languages. Elovitz and those who followed her work hardcoded these categories into their text-to-phonetics translators.

The RAVE text-to-phonetics translator also keeps track of the translation process and handles special rules statements needed to create orthophonetic animation (like that described in Talking Tiles, where the letters glow at the appropriate times). Needed information is generated and stored in an Orthophonetic Correspondence Record (OCREC) that tells which orthographic characters were associated with which phocodes, and what the orthographic context was examined to determine that pronunciation.

The number of rules needed depends on the complexity of the human language.[7] A language like Russian that is relatively consistent phonetically basically requires dozens of rules. Spanish and Greek are rather easy, too (see Figure 3, this page). But rules for incorrigibly inconsistent languages like English run to hundreds of statements, plus individual exceptions. Each exception is defined by its own rule statement and optional custom orthophonetic rules to go with it.

Often difficult choices must be made about how to animate these quirky exceptions. Take the word "one." The RAVEL coder could decide to write orthophonetic rules to state that all three letters are to be displayed for any part of, and for the whole word, with no other effects generated. This would reflect a decision to indicate to the viewer that the word "one" cannot be decomposed into sounds that match the letters in any logical manner. Another possibility would be to indicate that the "o" of "one" made the sounds "w" and "uh," the "n" made the sound "nn," and the "e" was silent.

RAVEL source-code orthophonetic rules empower the designer to make these decisions and have them be transferable to other host machines rather than having them built in with low-level programming. These rules enable the application designer to create synchronized orthophonetic animation at any level, from having words light up as they're read, to having individual letters light

(continued from page 26)

up as a word is sounded out. You could even have an automatic Mitch Miller follow-the-bouncing ball.

The orthophonetic rules include various effects parameters that are set up to be as general as possible. They may specify the offset within the source string at which the animation is to occur. This enables animation of symbol-set combination modes in which characters cause sounds to occur in an order different from that in which the characters are arranged (Pig Latin, for instance). Other possibilities may involve overlapping elements of symbols (Oriental ideographic characters); denoting particular modes of effects; or specifying synthetic actor commentary to be associated with that rule ("this vowel is long because of that silent e"). Unfortunately, a detailed description of these considerations cannot be presented in this article.

Coordinating the Action

At run-time, RAVE routines rigorously coordinate the presentation of the images, sounds, and symbols to create the illusion of life. Each time the synthetic actor is called upon to speak, RAVE software executes a two-phase process. Phase one prepares optimized intermediate data in a form that enables maximum execution speed during the real-time phase two. As much of the processing as possible is completed prior to the commencement of actual speech and animation. This is accomplished by generating data structures of pointers and lists (called "scripts") for the real-time processes.

When direct to pronounce a phocoded string or word, the RAVE NDI translates it and generates an appropriate audio script for the narrator device that produces that synthetic actor's voice. RAVE routines extract necessary characteristics of the synthetic actor from the RAVEL program and from other sources to create this and the graphic animation scripts.

The animation processors that run these scripts in real time may be invoked parametrically, or to speed up the real-time phase, the RAVE system may actually generate these scripts itself by using internal compilation during phase one. When scripts are complete for the voice and animation of the synthetic actor, the RAVE real-time coordinator takes over. The coordinator cues and handles events and timing interrupts.

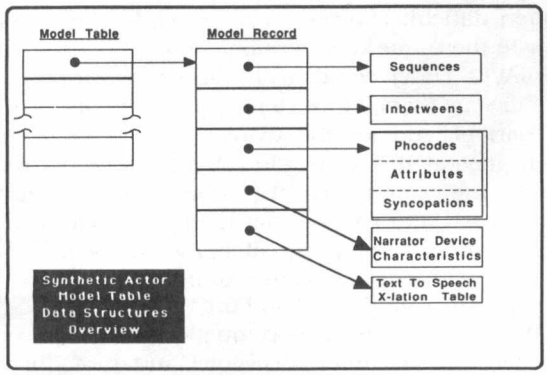


Figure 4: *RAVE/RAVEL Data structures overview. Each synthetic actor has an entry in the model table. The actor's model record contains pointers to information structures that define its characteristics.*

Ideally, the timing is coordinated with feedback from the narrator device driver. For instance, the device driver may tell which phoneme of the audio script was being pronounced at a particular time. For narrator device drivers (such as Macintalk) that do not provide such feedback, timing parameters may be associated with each image-animation frame. They may also be used in the case where feedback is available only at a higher level of granularity than is required for the animation sequence. For instance, many English speech synthesizers have only a single phonetic segment code signal for certain diphthongs. Yet, the sound is best displayed by two articulatory positions.

The behavior of the synthetic actor not associated with actual speech (for instance, when one turns toward the user or blinks) is controlled by a special behavior controller in RAVE and implemented with sequences of images just as the speech segments are. Each action or behavioral trait is given its own coded representation. To create a unique personality for each synthetic actor, a neural-net simulator is driven with RAVEL statements that define low-level behaviors. The intention is to create a software platform extensible to full modeling of human form and behavior.

RAVE Database Structures

At the heart of the database structures compiled from RAVEL source code is a dynamically allocated synthetic actor model table. The table structure provides for specification and simultaneous control of multiple actors, each with its own appearance, behavior, and language. Each record contains a set of pointers to variable-length tables that define particular information about that actor (Figure 4, this page).

The first pointer indicates a sequences table that defines images and synchronization characteristics for display of the synthetic actor. Next, a list of in-betweens defines intermediate images and the parameters that govern their insertions. In betweens can be specified in RAVEL, nested to any depth, and associated with specific timing intervals or dependent on the timing of context images.

The phocodes table defines the narrator device characteristics of the synthetic actor in terms of its speech segment and other codes. Each code has its own record in which a field specifies the number of bits in that particular code and a definition of it. This enables the calling application to manipulate phonetic encodings without knowing how the particular narrator device functions. An associated syncopations table describes the phonetic codes necessary *to sound out* words in unblended speech.

A table of attributes is also associated with the phocode table. It includes a flag that designates which phocodes are event phocodes (i.e., those for which explicit feedback is available from the narrator device). Another flag designates the orthophonetically significant phocodes (i.e., those that matter when creating and

(continued from page 28)

interpreting OCRECs). An intonation field indicates phocodes associated with stress or other intonation that may influence adjacent phocode timing. Another field basically indicates whether the phocode represents a vowel sound. This can be utilized in certain text-to-phonetic conversion methods for assignment of intonation. Knowing which phocodes are vowel sounds is necessary for syllabication to determine stress assignment and prosody in almost all languages.

Another pointer leads to a table of narrator-device characteristics stored in a format convenient to the audio script processor. This usually includes values describing speed, pitch, volume, pause codes, and narrator-device calling sequence idiosyncrasies. The final pointer in this simplified diagram contains the address of the text-to-speech translation table for the synthetic actor. It defines special codes and contains the rules stored in compressed concatenated fashion.

Other Features

Randomness in the behavior of synthetic actors is provided for through the use of RAVEL constructions that enable the designer to specify a set of images rather than a single image for display. Which image is actually used is determined randomly at run time. This enhances the illusion of life (living things are a bit unpredictable and just don't repeat motions exactly the same each time). The submodel architecture for defining particular RAVE actors can handle such problems as turning, tilting, and nodding of the head during speech (real people usually don't hold still as they talk). Extensions of RAVEL will eventually let such behaviors be programmatically coordinated with syntactic or even semantic components of elocution.

The RAVE system allows for anticipation or leading the audio by the motion. This can enhance the lip-sync perception[8] apparently through suggestion of causality, or perhaps by simulating the fact that you see images slightly before you hear sounds in real life (because of the faster speed of light). Adjustments for intonation (for instance, making the mouth open wider on a stressed syllable) are also provided for in the design.

For the sake of simplicity, the use of RAVE and RAVEL software has been described as though each articulatory position were invariant with a phonetic output. In real life, lip positions vary in normal speech, depending on context. For instance, the usual position of the lips as they perch to deliver the second "b" sound in "beebee" is noticeably different from the position of the same "b" in "booboo" where they are slightly protruded in anticipation of the "oo" sound. Detailed descriptions of the RAVE mechanisms that handle these context-dependent coarticulatory variations (as well as other advanced HyperAnimation features) are beyond the scope of this article.

Other possibilities besides a talking head can be created by using HyperAnimation software to coordinate sound and motion. Examples include a beating heart for teaching medical students, moving hand gestures for sign language, or fingering positions for a musical instrument. These possibilities rely on the fact that RAVE also works with digitized sounds, and not just synthetic speech. Another implication is that synthetic actor applications can be prototyped by using synthetic speech that is then replaced by custom-made prerecorded digitized speech in the production version (this is how Alphabet Blocks was developed).

We have used a number of studies, especially in the matter of sequential synchronization through recognition of digitized speech.[9,10,11] We are now building on these studies, particularly in order to enhance the efficiency of integrated HyperAnimation tools in working with the sort of prerecorded digitized sounds mentioned previously. The recent appearance of a superb system for capture and manipulation of sounds on the Macintosh (Farallon's MacRecorder[12]) makes this a fruitful area for further work.

The HyperAnimator

We are currently field-testing the HyperAnimator development system, which will enable developers to use a device-driver version of RAVE to direct synthetic actors from standard applications programs. Significant additional effort was expended to make its calling sequence compatible with Macintalk. That means no new bindings are needed. Not only can you prototype with Macintalk alone to save time, but existing programs that now use Macintalk can get a "facelift" by adding a synthetic actor.

Not only is the device driver currently running on the Mac, but also a HyperCard XCMD interface that enables the Hypertalk programmer to call synthetic actors into HyperCard stacks. Embedded escape sequences within the text parameter passed by a single Hypertalk verb control the position, appearance, and disappearance of synthetic actors. Using this method instead of multiple verbs enables it to be compatible with the Macintalk calling sequence, simplifies the Hypertalk binding, and avoids the problem of device-driver protocol variance on other machines. The design goal was to minimize porting difficulties by providing a simple interface for invoking the device driver that will be available in multiple operating environments.

As an example of how HyperAnimation technology can be used in HyperCard, HyperTalk source code (Listing One, page 66) was presented for part of an interactive stack demonstrated at January's MacWorld Expo. This code features a synthetic actor reading a poem as it scrolls up a line at a time (Figure 5, page 34). Note that the user can rate the recital by pressing one of two buttons.

The HyperCard synthetic actor lives in its own little window, (in this case, one which has no frame). You can click anywhere on the face window and drag it to a different place on the screen, or control it from Hyperddtalk with the MOVE command. That and other control sequences are passed in the text parameter of the XCMD verb "RAVE" by bracketing them within the ¦~ and ~¦ escape-code sequences as shown.

Future Fun

As of this writing, RAVE software has been released only

for the Macintosh. But the HyperAnimation tools are quite portable. RAVE was written in C using a shell that isolates the code from its operating environment. Early development work (before the Macintosh was released) was done on the IBM PC. Bright Star has also performed prototype work on the Amiga. From the beginning, the development team at Bright Star has sought maximum portability of the system by preparing for portability with macros and coding conventions set up after careful consideration of potential host environments. Bright Star intends to license the HyperAnimation technology (patents still pending) for special-purpose dedicated systems.

But first, HyperAnimation software will be placed in the hands of the Macintosh community. That machine will become the initial laboratory for mass experimentation with synthetic actors much in the same way it did for fonts, clickart, and desktop publishing. We expect to see a vanguard of applications and synthetic actor models (both public domain and proprietary) developed and used there in the first wave of a creative explosion caused by the availability of this newest and most intuitive generation of human interface.

HyperAnimation technology eventually will have a pervasive influence on diverse fields. In many ways, though, the most important use of synthetic actors will be in education. Bright Star has taken great care to create early educational applications of HyperAnimation that are innovative, fun, and educationally sound. But Alphabet Blocks, Talking Tiles, and other educational products using HyperAnimation are barely the beginning. They offer only a taste of what software using this technology will someday be capable of doing. Particularly in literacy education, the language independence of the HyperAnimation platform will eventually ensure that as computational capabilities progress and prices decline, teaching tools of unprecedented power will help bring learning to all the peoples of the world. With simulated teachers to help, human enlightenment can become a less linear function of human resources. Face it—we need it.

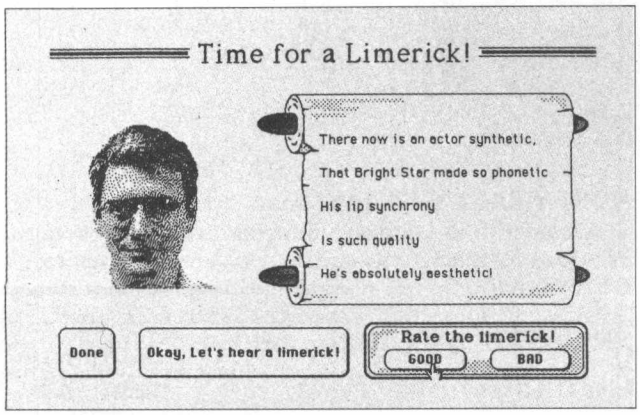


Figure 5: *Bright Star's HyperAnimator enables stackware designers to use lip-synced talking synthetic actors themselves.*

Availability

For those interested, Bright Star Technology publishes The HyperAnimation News. For a free copy, write to: Bright Star Technology Inc., 14450 NE 29th St., Bellevue, WA 98007.

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Bibliography

1. Sculley, John, et al. "Knowledge Navigator" and "HyperCard Future." Video presentations, Apple Computer, 1988.
2. Potmesil, M. and Chakravarty, I. "Modeling Motion Blur in Computer-generated Images." Proceedings, SIGGRAPH '83, Computer Graphics, 17(3):389-399, 1983.
3. Parke, F.I. "Parametrized Models for Facial Animation." IEEE Computer Graphics and Applications, 2(9):61-68, 1982.
4. Witten, I. "Principles of Computer Speech." London: Academic Press, 1982.
5. Klatt, Dennis H. "Review of text to speech conversion for English." Journal of the Acoustical Society of America, 82(3):737-793, 1987.
6. Elovitz, Honey Sue; Johnson R.; McHugh, A; and Shore, J.E. "Automatic Translation of English Text to Phonetics by Means of Letter to Sound Rules." United States Naval Research Laboratory Report 7948, December, 1976.
7. Sherwood, Bruce. "Fast Text-to-Speech Algorithms for Esperanto, Spanish, Italian, Russian and English." International Journal of Man-Machine Studies, 10:669-692, 1978.
8. Thomas, Frank and Johnston, Ollie, "Disney Animation: The Illusion of Life." Walt Disney Productions, 1981.
9. Bagley, J.D. and Gracer, F. "Method for Computer Animation of Lip Movements." IBM Technical Disclosure Bulletin, 14(10), 1972.
10. Bakis, R. "Speech Recognition System." IBM Technical Disclosure Bulletin, 13(4), 1970.
11. Magnenat-Thalmann, Nadia and Thallman, Daniel. "Computer Animation: Theory and Practice." Chapter 9: Human Modeling and Animation, Springer-Verlag, 1985.
12. MacRecorder Sound System, Farallon Computing Inc., Berkeley, CA, 1988.

HyperAnimation, HyperAnimator, RAVE, RAVEL, Alphabet Blocks, and Talking Tiles are trademarks of Bright Star Technology Inc.

DDJ

(Listing begins on page 66)

Vote for your favorite feature/article.
Circle Reader Service Card **No. 1**

SQL DEVELOPMENT TOOLS

Leveraging the PC's increased processing power and improved graphical interface to streamline program development.

by Michael Geary

Although the structured query language (SQL) has been around for a number of years, only recently have programmers familiar with the microcomputer environment had to come to grips with it. This is due primarily to the momentum generated by the more powerful hardware platforms—specifically the 80386 and the 68020—that are being used with high-performance PCs. SQL demands a lot of computing horsepower, and it has only been recently that PCs have been potent enough to drive SQL-compatible applications. More powerful hardware, of course, enables more powerful software, and IBM's planned introduction this month of the OS/2 Extended Edition, an operating system that will include an SQL database engine, is also significant. Such an engine will enable programmers to develop database applications that run across dissimilar computer environments, from PCs to minis to mainframes. At the same time, users can efficiently and transparently tap into databases using SQL-compatible application software—Paradox, dBase IV, Ingres, or custom databases, for instance—that acts as a front-end to the SQL database engine on the server.

Michael Geary is a software designer at Gupta Technologies Inc. He can be reached via e-mail BIX or GENIE where he goes under the name of "Geary" or by regular mail at P.O. Box 1479, Los Gatos, CA 95031.

SQL consists of only 12 keywords which perform the standard database operations of data manipulation, definition, and control. Because the language (actually a sublanguage) is standardized, a number of development tools that make it simpler to link application program front-ends with SQL server engines have become available. These development systems typically fall into one of two categories: 1. traditional command-line programming tools that have a programmatic interface and use preprocessors or precompilers to translate embedded SQL commands into the host language (see the accompanied sidebar); and 2. interactive graphical tools that make it easier for programmers to construct forms, report writers, tables, and other database features. SQLWindows, a database application development system for Microsoft Windows, which I developed for Gupta Technologies and will describe in this article, has aspects of both.

Applications built with SQLWindows are true Windows applications with the full Windows user interface: scrollable table and form windows with pull-down menus, dialog boxes, pushbuttons, radio buttons, and other sorts of gadgets. SQLWindows applications can access Gupta Technologies' SQLBase and IBM's DB/2. Future releases will be able to use other databases, such as the SQL Server from Sybase, and will run on Presentation Manager and the Macintosh as well as Windows.

Figure 1, next page, shows three SQLWindows applications in action: a database of SQLWindows bugs that is used internally to develop the tool, a small window showing part of the SysColumns table, and the SQLWindows calculator. Although SQLWindows is mainly designed for writing database applications, it can be used to construct other kinds of Windows applications, such as this calculator.

Anatomy of an SQLWindows Application

SQLWindows provides the developer with several tools that work together to construct an application:

- A window editor for forms, tables, and dialog boxes. This lets you design and edit the visual aspects of the application.
- A procedural language, SAL (SQLWindows Application Language), for coding the actions to take place on menu clicks or other events.
- A structured outline editor, for editing the application outline. A SQLWindows application is defined by an outline that contains all the form layout descriptions, data declarations, and SAL code needed to execute it. The outline editor is used

both for entering SAL code and declarations and for setting optional choices, such as whether a field should have a border.

The outline editor is tied directly to the window editor. Since the application outline is the complete description of the application, every change you make with the window editor, changing a window size, such as is reflected in the outline. Similarly, if you edit the outline directly, any change that affects the appearance of a window causes that window to be redisplayed appropriately. To make it easy to find your way around the application outline, if you click the mouse on a field or menu item in a form window, the outline opens up and scrolls to make the correct outline entry visible.

Figure 2, this page, shows the SQLWindows outliner with the calculator application open. The outliner is zoomed up to full screen in this view, with the calculator window visible on the right. I selectively expanded and collapsed portions of the outline to give an overview of the various sections of an SQLWindows application outline. In the SQLWindows outliner, a solid diamond indicates that an item has additional items nested within it (which may or may not be visible); an open diamond indicates the item has no additional levels.

One of the nice things about using an outliner is this ability to focus in on what you are working on and to collapse out other portions. In fact, the main reason I chose an outliner as the base for SQLWindows is the way it ties everything together cleanly. In similar systems I've seen on the Macintosh, a series of dialog boxes are used for the various settings associated with each field. This is easy to work with, but doesn't give you any way to get the "big picture"—you can only look at one or another of these dialog boxes at a time. The outliner gives more flexibility; you can deal with the details of any particular item without distraction, and you can also get an overview of how your application is taking shape.

At the top of the outline are some global declarations, including references to the external libraries SQLWIN.EXE (SQLWindows itself) and USER.EXE (part of the Windows system). An SQLWindows application can call external functions in any Windows dynamic link library, thus extending the SQLWindows system. These external functions can be written in any language that can produce a Windows dynamic link library. The Global Declarations section also includes variable and constant definitions, along with internal functions written in SAL.

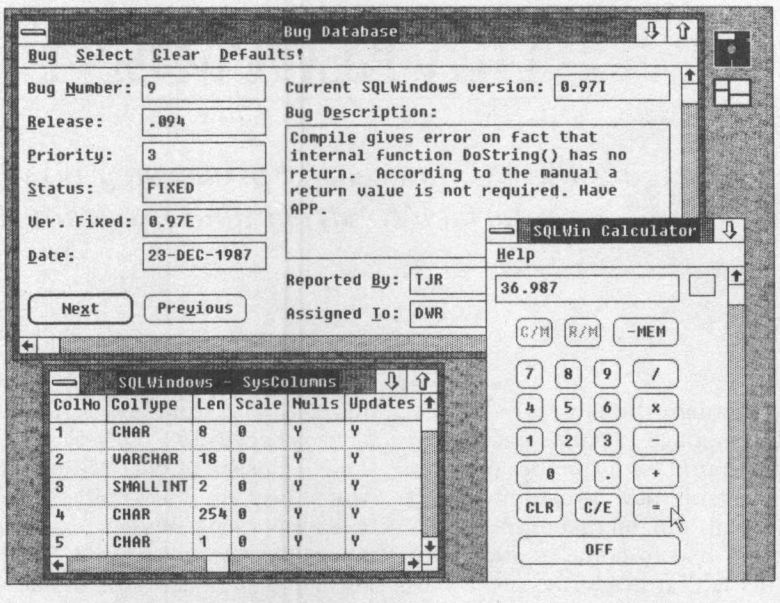


Figure 1: *SQLWindows in action, running three applications. The large window at the top is the Bug Database we use to track SQLWindows bugs. Below that is a small table window showing a portion of the SysColumns table. The SQLWindows calculator—a popup calculator program written in SQLWindows—is at the lower right.*

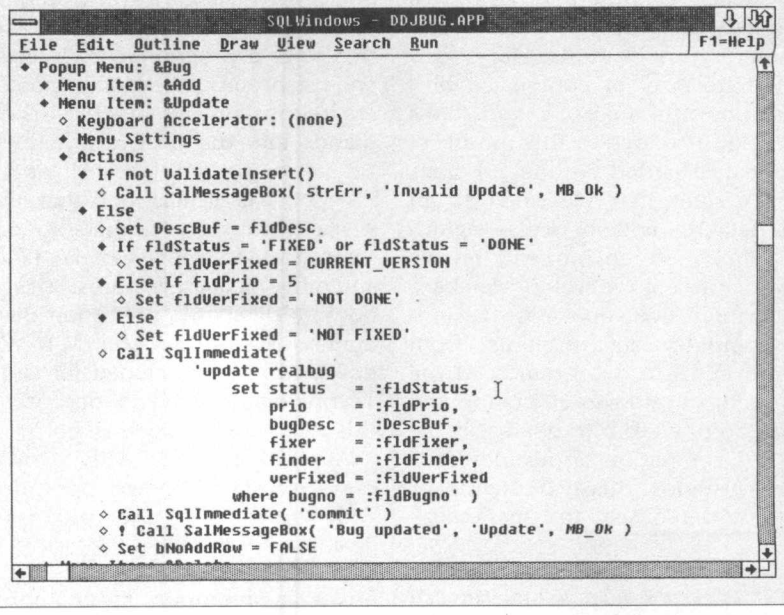


Figure 2: *Coding an SQLWindows application. The outline window shows the window definition and SAL code that implement the R/M (Recall from Memory) button in the SQLWindows calculator application. The R/M button has code for two messages: SAM_Create and SAM_Click.*

(continued from page 37)

The second major section of this outline describes the calculator window itself, declared as a form window. (This "form" just happens to contain mostly pushbuttons rather than data fields!) The Menu section shows the SAL code that implements the Help menu item in the calculator window. It simply calls the *SalCreate* Window function to bring up the *dlgHelp* dialog window. (You can see the first line of the definition of *dlgHelp* at the bottom of the outline.) The Message Actions section contains SAL code to process any messages this form window needs to handle—in this case, the SAM_Create message, which notifies the window that it has been created.

Messages are one of the greater mysteries of Windows programming, and one of the causes of the fabled learning curve faced by programmers who are new to Windows. Messages are also one of the things that give Windows its power. SQLWindows doesn't try to hide this message-passing architecture; an SQLWindows developer must deal explicitly with messages. So, we had better take a look at just what they are.

Windows and Messages

An SQLWindows application, like any other Windows application, is built around a collection of various kinds of windows. Nearly every visible object on the screen is a window. In the calculator program, the form itself—with the title bar, and so forth—is a window. The items inside the form, which are mostly pushbuttons in this particular form, are also individual windows. This is important because all these windows can receive messages and execute program code based on what messages they receive. Every Windows application works this way; each is a collection of windows, each of which has executable code that responds to various messages.

What are messages? For the most part, they are notifications from SQLWindows or from Windows itself that some event has occurred. Each message has a name, which is really just a defined numeric constant that identifies the message. Figure 3, page 39, shows some examples.

When a window receives one of these messages, it may perform any actions needed, by executing procedural code. For example, a form window may initialize its data fields on the SAM_Create message. A window may disregard any message if no special processing is needed for that message. A pushbutton, say, might have code for a SAM_Create and a SAM_Click but not for a SAM_Destroy. (SAM_, by the way, stands for SQLWindows Application Message.)

Newcomers to Windows (and Macintosh) sometimes have a little trouble with this business of receiving messages. It's a little backwards compared to traditional programming, where the program is always in control and makes explicit requests for user input. Here, the program is a collection of pieces of code that are called by the system in response to certain events, including user input. The program doesn't know that order in which these events will occur; it just has to be ready to respond to them at any time. (There is some predictability—the very first message to any window will be SAM_Create and the very last will be SAM_Destroy.)

Although this architecture seems foreign to people at first, it has a big payoff: the modeless user interface that Windows and Mac applications feature. Instead of having menu

SAM_Create	Sent to each window when it is created
SAM_Destroy	Sent to each window when it is destroyed (closed)
SAM_Click	Sent to a pushbutton when the user clicks the button with the mouse or keyboard
SAM_SetFocus	Sent to a data field when the user selects the field for editing by tabbing to it or clicking in it

Figure 3: *An example of messages that may appear after an event has occurred.*

SQL
(continued from 39)

mode, edit mode, insert mode, and who knows how many other modes, a Windows application—if put together right—can avoid most modes entirely. The user doesn't get locked into one corner of the application or another, but has all its facilities available.

Constructing an SQLWindows Application

Long before any procedural code has to be written, the SQLWindows developer can put together the basic shell of the application by laying out

Embedded SQL Development Kits

One tactic programmers can take when writing SQL-compatible custom database applications with high-level programming languages is to embed English-like SQL commands directly into a program. (SQL was, in fact, originally designed with constructs to facilitate embedded coding.) To do so, however, programmers must generally use developer's toolkits that include preprocessors or precompilers in order to convert SQL-executable statements found in an input file to appropriate calls in an output file. That output file can subsequently be compiled, linked, and executed as normal.

```
SQL BEGIN DECLARE . . .
   char host[20] = ' d:750vms-t:
   char host[20] = ' d:750vms-t:
E SQL END DECLARE . . .
   EXEC SQL DECLARE ORA1
                          DATABASE;
   EXEC SQL DECLARE DB2
                          DATABASE;
EXEC SQL DECLARE CONNECT
:usr IDENTIFIED BY :pwd
      AT ORA1 USING : host1;
      AT DB2 USING : host2;
```

Example 1: *The precompiler translates statements into the appropriate host language.*

```
#include<stdio.h>
$include sqlca;
#define SQL_ERROR sqlca.sqlcode
#define SUCCESS O
raise_managers(raise)
$double raise;
{
$database company
  if(SQL_ERROR)return(SQL_ERROR);
  $update personnel
    set salary = salary*(1.0+$raise)
      where(year(hire_date)<1985)and
        exists(select manager from
                              offices
          where manager=
                          pers_num);
  if(SQL_ERROR)return(SQL_ERROR);
  return(SUCCESS);
  }
```

Example 2: *Illustration how ESQL can be used.*

The overall advantage of this approach is that SQL, which is a nonprocedural language, can take on some of the constructs of a procedural language such as C, Fortran, Pascal, Basic, and so on. Among other things, this means that embedded-SQL programming is more conceptually oriented and, consequently, easier to understand than programming in SQL only. It also means that a single program can be used with data in several different databases. In any event, the resulting applications tend to be more flexible than applications written only in SQL or only in a high-level language.

Although development time can often be shortened because less code needs to be written for file or database access and manipulation, an extra step or two is included in the development process. In a conventional scenario, for instance, you might write a C program, compile it into an output object file, link and edit the object file to create an EXEC file, and then run the program. With an embedded-SQL command program, however, you would: write the program, precompile it into an output file, compile the program into an object file, link-edit the object file into an EXEC file, and finally run it.

A number of toolkits that are compatible with most conventional high-level languages, including C, Cobol, Ada, Fortran, Basic, and Pascal, are available from different developers. Developer toolkits such as these solve a number of basic database creation problems, the most basic problem being that SQL is a nonprocedural language and, as such, is limited. In addition to performing complex data manipulation routines, embedded SQL commands let you create and alter databases and tables, retrieve and modify data, and add indexes. They also translate datatypes into the host language and automatically generate the code to do the conversion.

The Pro* series from Oracle is one such embedded-SQL development toolkit. Currently, the series is available in a number of implementations that support C, Pascal, Fortran, Cobol, PL/1, and Ada. With the Oracle kits, all SQL statements are prefixed with the words EXEC SQL. When run through the precompiler, all statements beginning with EXEC SQL are translated into the appropriate host language. See Example 1.

In this example, ORA1 and DB2 are logical names used by the connect statements. The USING clause specifies the network, machine, and database to be associated with the name ORA1 or DB2. As with cursor and statement names, *ORA1 and* DB2 are not host variables, but identification names given for ease of use.

The ESQL-family from Informix is another development tools series. Versions of the kit are currently available for C, Cobol, and Ada developers. In the Ada and Cobol versions of ESQL, an EXEC SQL statement placed at the beginning of the line tells ESQL that the following line or lines should be processed as ESQL statements. With C, a dollarsign ($) is used. The C code in Example 2 illustrates how ESQL can be used.

When used with an ESQL statement, the $ implies that the immediately following identifier is a C variable, and not the name of an SQL column or table. This code also illustrates a subquery condition associated with a single field condition. —eds.

the various windows and dialog boxes. This is all done with a simple visual window editor, similar to an object-oriented "draw" program such as Windows Draw from Micrografx Inc. You can lay down items in a form, move them around and resize them, give them titles, set up the menu bar and popup menus for a window, and do it all using the window editor and the outline editor. It's quite practical for one person to prototype the visual appearance of an application this way, then hand off the application outline to someone else to write the detailed code.

Normal Windows editing facilities—cut, copy, and paste—are available during application development, which makes it easy to duplicate items in a form or copy over portions of one application into another. As mentioned before, the window editor and outline editor are tied together, so the outline always reflects the current application structure.

When it does come time to start adding procedural code and SQL calls, the outline editor is used to enter SAL code, which looks much like any block-structured language, such as C or Pascal. The biggest difference is that the code is entered at specific points in the outline, and the code itself is indented in outline format. Unlike C or Pascal, this indentation is significant; the "dependent" statements of an *If* statement, for example, are those statements nested under the *If* statement in the outline. This eliminates the need for curly braces or BEGIN END.

Figure 4, page 44, shows a portion of the application outline for the Bug Database from Figure 1. This section of code implements the *update* command in the Bug menu. It performs some field validation, sets the fldVerFixed status indicator, and calls SQL to update the database.

One handy feature in the SAL code is that you can refer directly to any item in a form by name, and SQLWindows will fetch or set that field value in the actual window contents. Fields work just like variables in this regard; you don't have to make special Windows calls to deal with their values. In Figure 4, for example, the SQL "update" statement takes its values such as fldStatus directly from the form fields and updates the database with those values.

External Libraries and DDE

SQLWindows is designed so that most common database applications can be written completely using the SAL language and other features of SQLWindows. For applications that push the limits, it's easy to call dynamic link libraries, as mentioned earlier. This approach lets you write the bulk of your application without having to deal with the C compiler and the Windows Software Development Kit, and yet use those tools when necessary to write those special functions that you need. You can also call existing dynamic link libraries such as the graphics library from Micrografx. Calls to these external functions are coded just as any other SAL function call; it's necessary only to declare the function and library in the External Functions section of the outline.

SQLWindows can also talk to other Windows applications through the clipboard and Dynamic Data Exchange (DDE). DDE lets you set up cooperating applications that communicate with each other under program control. For example, you can write an Excel macro that passes data automatically from a spreadsheet into a SQL database through a SQLWindows application. Or, using the Proteus communications program, you can download data into a SQL database from an on-line database service, and then view the latest information through a SQLWindows form.

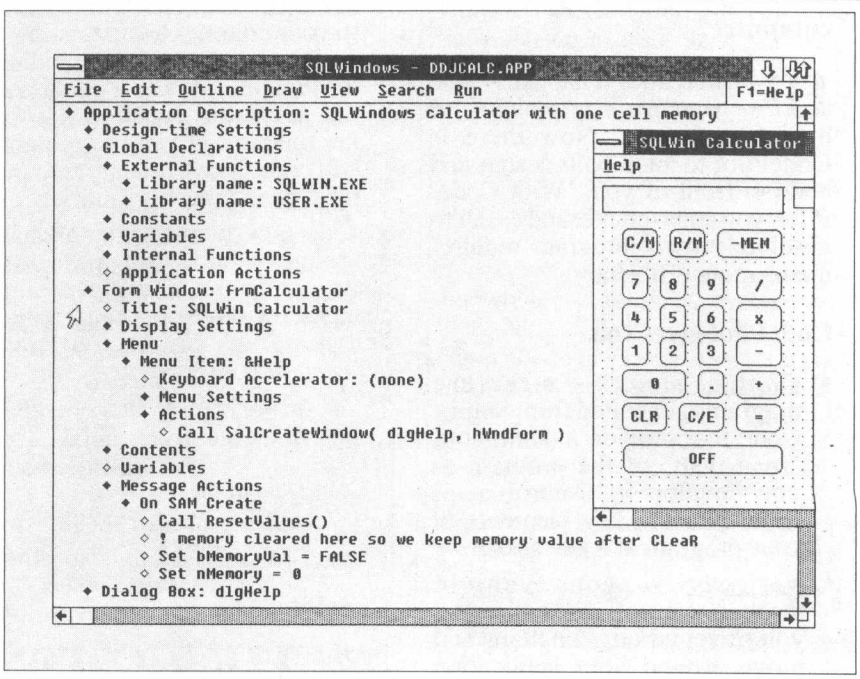


Figure 4: *Some detailed SAL/SQL code. This is the SAL code for the Update menu item under the Bug menu in the Database from Figure 1.*

Testing and Debugging

For debugging, SQLWindows provides several facilities including breakpoints with user-defined break actions and an animated program trace. It's also very easy to add a temporary test field in a form, or create another window, to display debugging information directly from your SAL code. A simple Set statement will display a new value in a field, making this approach handy.

Conclusion

SQLWindows attempts to combine the intuitive appeal of the Windows user interface with the power of a SQL database. It's designed for database application developers who

want to build Windows applications without the effort of using the Windows Software Development Kit and C compiler.

SQLWindows does not yet have any automatic form generation, which would produce a default form given the name of a database table. Nor does it have graphic Metaphor-like tools for setting up connections between tables. Instead, the developer writes explicit SQL code to access the database. These other goodies will be coming in future releases, but for now we have tried to make sure that the developer can accomplish the basic task of quickly producing database applications that provide the full Windows user interface.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Database Research—Is There Life Beyond?

Researchers at the University of California at Berkeley aren't content with current database technology. While acknowledging that the relational model has solved many of the problems presented by rigidly structured data, a team of programmers led by Professors Michael Stonebraker and Lawrence Rowe also believe that there are categories of information where database technology can be applied to less structured, more dynamic data. Such complex data might include documents, engineering data, geographical data, CAD/CAM data, and programs. In these instances, Stonebraker thinks, databases might be represented more easily by a collection of rules rather than by data.

"Current database systems are great for doing data processing type things," says Stonebraker, "but they tend not to work when you stray outside of the norm. If you want to store the layout of a building or of a computer chip, for instance, they don't work well at all. That's why our objective is to be able to do object management as well as data management."

The database research at UC Berkeley revolves around two core programs: the development of a new database management system called Postgres, and the development of a supporting application development environment called Object FADS. Postgres (which stands for "after Ingres," Ingres being a DBMS project initiated in the early 1970s), is a rule-based relational database system whereby the user can define rules that provide inference mechanisms alerters, triggers, and other pertinent rules. These rules may follow either a backward- or forward-chaining control paradigm, depending on which is more appropriate for a specific application; the rule subsystem itself can decide which is more suitable based on statistics about the current database use. A priority mechanism is also provided, so that if many rules apply at a certain case, the one with the highest priority will be used.

The project will also include the design and analysis of algorithms that support derived data objects (or views) in relational DBMSs, as well as a collection of rule-indexing algorithms, several lock-based algorithms, and a "new generation" of query optimization techniques that deal with rule processing, semantic optimization, and multi-statement optimization.

Among the many unique algorithms the team has formulated is one that Stonebraker describes as an "asynchronous vacuum cleaner." Instead of immediately deleting or creating an audit trail of data as it is updated, the algorithm will simply add the new data to the system. At the same time, the system keeps track of the data that has been obsoleted and periodically collects, then discards the data that hasn't been used for a specified period of time.

The other component of the project, Object FADS, is an object-oriented development environment for interactive database applications that use a WYSIWYG interface. This project is an offshoot of an earlier system called Forms Application Development System (or FADS for short). Object FADS uses a program interface called a "portal" which allows blocks of data specified by a query to be retrieved and updated randomly. The prototype of the Object FADS system is being written in Common Lisp and has been implemented on the X/Common Lisp Interface under X-Windows.

Object FADS uses a shared object hierarchy programming environment. Since this environment is an extension of the Common Lisp Object Standard, shared classes are possible. The Object FADS toolkit includes an object-oriented forms system for building user interfaces that allows arbitrary nesting of field types. A shared-object browser, which allows programmers to browse through stored objects, is also being developed, along with a visual form editor and visual object editor.

To test the Postgres approach, the development team is using the system to create an integrated knowledge base that will analyze the water drainage system of portions of California's Central Valley. Initially, several sets of depth-to-groundwater measurements will be entered into Postgres using a customized spatial access method. The data sets will be of differing granularities, obtained at sporadic time intervals, and contain inconsistencies. The overall goal of the application is to provide a knowledge-based system that can integrate loosely related data sets so that users can make queries in a transparent manner. "Our objective here," explained Stonebraker, "is to steal a piece of what expert system shells do and do it better inside the database system."

Stonebraker believes that the biggest advantage to programmers of a DBMS like Postgres is that "they won't have to figure out how to get at islands of data since both objects and data will be managed in the same way." —eds.

PATTERN MATCHING:

THE GESTALT APPROACH

An introduction to the Ratcliff/Obershelp pattern-recognition algorithm.

by John W. Ratcliff and
David E. Metzener

What is the gestalt approach to pattern matching? *Gestalt* is a word that describes how people can recognize a pattern as a functional unit that has properties not derivable by summation of its parts. For example, a person can recognize a picture in a connect-the-dots puzzle before finishing or even beginning it. This process of filling in the missing parts by comparing what is known to previous observations is called *gestalt*.

The Ratcliff/Obershelp pattern-matching algorithm uses this same process to decide how similar two one-dimensional patterns are. Since text strings are one dimensional, this algorithm returns a value that you can use as a confidence factor, or percentage, showing how alike any two strings are.

Because this pattern-matching algorithm can recognize matches in substrings quickly and easily, there are many applications for it. For example, a compiler using this algorithm would be able to determine what variable, keyword, or procedure name the programmer meant, even when the compiler encounters a spelling error. Educational software that can recognize a correct answer contextually (even when the answer contains a typing error) is another natural application. A command shell could finally recognize that SYMPONY doesn't exist—and do something intelligent with that information, such as pop up a menu of close alternatives like SYMPHONY. Text adventure games with their powerful parsers are an ideal application for this algorithm: the games could make broad assumptions in assimilating user input.

The Ratcliff/Obershelp pattern-matching algorithm was developed by John W. Ratcliff and John A. Obershelp in 1983 to address concerns about educational software. Often, educational software has consisted of multiple-choice questions only because the existing algorithms required an exact character-for-character match. The algorithm presented in this article is both forgiving and understanding of simple typing mistakes, and allows intelligent responses to erroneous input. To date, this algorithm has been implemented in a commercial spelling checker, a database search program, and a compiler.

Adding this algorithm to a compiler had some dramatic results. When this algorithm was implemented in a primitive C compiler, the compiler was able to make accurate assumptions when it encountered misspelled procedure names, keywords, and variables. When it couldn't find an identifier, it examined all of the currently defined names and collated the best matches. If the compiler could find no match better than 60 percent, then it produced a normal error message. The most common case, however, resulted in an accurate and unambiguous match: the compiler was able to continue with this assumption while producing both a warning message that indicated the assumption made and the line number that it occurred on. The result was that rather than getting a cascade of 50 warning messages because of one typing mistake the programmer now got a simple warning message and a successful compilation. After finding several strong matches, the compiler prompted the programmer for confirmation in a pop-up window. The compiler could even go so far as to ask the programmer if it should automatically correct the source code as well. On the occasions when the compiler made a false assumption, it almost always generated errors due to mismatched arguments being passed to an assumed procedure. Even if an erroneous assumption results in a successful compilation, the programmer is still warned and

John Ratcliff has developed a number of educational packages at Milliken Publishing, most notably the Word Math series, and is currently doing cardiovascular research at St. Louis University and is developing a computer game for Electronic Arts.

David Metzener is currently a researcher in the cardiovascular division at the St. Louis University. He has been writing educational software applications since 1982.

knows not to run the executable that the compiler produced.

How the Algorithm Works

The best way to describe the Ratcliff/*Obershelp* pattern-matching algorithm, in using conventional computer terminology, is as a wild-card search that doesn't require wild cards. Instead, the algorithm creates its own wildcards, based on the closest matches found between the two strings. Specifically, the algorithm works by examining two strings passed to it and locating the largest group of characters in common. The algorithm uses this group of characters as an anchor between the two strings. The algorithm then places any group of characters found to the left or the right of this anchor on a stack for further examination. This procedure is repeated for all substrings on the stack until there is nothing left to examine. The algorithm calculates the score returned as twice the number of characters found in common divided by the total number of characters in the two strings; the score is returned as an integer, reflecting a percentage match.

For example, suppose you want to compare the similarity between the word 'Pennsylvania' and a mangled spelling as 'Pencilvaneya.' The largest common group of characters that the algorithm would find is 'lvan.' The two sub-groups remaining to the left are 'Pennsy' and 'Penci,' and to the right are 'ia' and 'eya.' The algorithm places both of these string sections on the stack to be examined, and advances the current score to eight, two times the number of characters found in common. The substrings 'ia' and 'eya' are next to come off of the stack and are then examined. The algorithm finds one character in common: 'a.' The score is advanced to ten. The substrings to the left—'i' and 'ey'—are placed on the stack, but then are *immediately* removed and determined to contain no character in common. Next, the algorithm pulls 'Pennsy' and 'Penci' off of the stack. The largest common substring found is 'Pen.' The algorithm advances the score by 6 so that it is now 16. There is nothing to the left of 'Pen,' but to the right are the substrings 'nsy' and 'ci,' which are pushed onto the stack. When the algorithm pulls off 'nsy' and 'ci' next, it finds no characters in common. The stack is now empty and the algorithm is ready to return the similarity value found. There was a score of 16 out of a total of 24. This result means that the two strings were 67 percent alike.

Inside the Code

Now that you know how the algorithm works, you're ready to look at the code. This article includes an assembly language routine that is accessible as a function call for C programs. (Listing One begins on page 68.) This assembly language routine has been optimized using techniques such as register optimization, algorithmic analysis, branch optimization, and instruction-cycle counts. Therefore, you may very well find this routine fast enough to be used as a basic string-comparison function in your software. In that regard you should note that the variables in this routine are declared as static, rather than dynamic, to make the source code easier to follow.

It should be clear from the earlier discussion that the time-critical portion of the code is in the section that determines the maximum number of characters in common between two substrings. The worst-case scenario is when absolutely no characters are found in common between the two strings. When this happens, $N \times M$ number of comparisons are required, where N is the number of characters in the first string and M is the number of characters in the second string.

The comparison procedure is composed of two loops: an inner loop for string two and an outer loop for string one. At each character in the two strings the procedure checks to see how many characters are equal. Whenever any characters are found that are equal, the procedure then checks to see if this number is greater than the previous maximum number of characters found. If it is, then the procedure updates the variable *maxchars* and updates the substring to be returned. Whenever a

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*************************************************************************/
/*                              GESTALT.C                                */
/*        written by John W. Ratcliff and David E. Metzener              */
/*                          November 10, 1987                            */
/*                                                                       */
/* Demonstrates the Ratcliff/Obershelp Pattern Recognition Algorithm     */
/* Link this with SIMIL.OBJ created from the SIMIL.ASM source file       */
/* The actual similiarity function is called as:                         */
/* int simil(char *str1,char *str2)                                      */
/* where str1 and str2 are the two strings you wish to know their        */
/* similiarity value.  simil returns a percentage match between         */
/* 0 and 100 percent.                                                    */
/*************************************************************************/
int     simil(char *str1,char *str2);
void    ucase(char *str);

main()
{
  char  str1[80];
  char  str2[80];
  int   prcnt;

printf("This program demonstrates the Ratcliff/Obershelp pattern\n");
printf("recognition algorithm.  Enter series of word pairs to\n");
printf("discover their similarity values.\n");
printf("Enter strings of 'END' and 'END' to exit.\n\n");
do
  {
    printf("Enter the two strings seperated by a space: ");
    scanf("%s %s",str1,str2);
    ucase(str1);
    ucase(str2);
    prcnt = simil(str1,str2);
    printf("%s and %s are %d\% alike.\n\n",str1,str2,prcnt);
  } while (strcmp(str1,"END"));
}

void ucase(str)
char *str;
{
while (*str)
  {
   *str=toupper(*str);
   str++;
  }
}
```

Example 1: *An example C program calling the gestalt functions*

new *maxchars* occurs you can shorten the search by the difference between the new *maxchars* and add that value. The reason for this is simply that once you have found, for example, a five-character match, you do not need to waste time looking more than five characters from the ends of the two substrings because there is clearly no chance of finding more than five characters. Inside the inner loop, whenever the procedure finds any characters in common, whether they form a new *maxchars* or not, the procedure advances the inner loop past these characters. On exit from this procedure, the DX register contains the number of characters found in common, and the variables CL1, CR1, CL2, and CR2 are pointing to the left and the right of the character string found in common between the two source strings.

The best way to explain this algorithm is . . . as a wild-card search that doesn't require wild cards.

The main procedure, *SIMIL*, which calls the *compare* routine, "realizes" there is nothing to place on the stack if no characters were found in common. If there are no characters to the left of either of the two substrings, then you don't need to push anything to the left. If there is exactly one character to the left of both substrings, you don't need to push the characters on the stack because they cannot be equal (or the first character would have been included in the maxchars substring). These same rules apply to the right of the substring as well.

Performance Aspects

To evaluate the performance of the routines, the following tests were performed. First, a series of strings were created from 1 to 20 characters in length. Then 10,000 calls were made to the pattern-matching procedure for each of these 20 strings on an 8 MHz IBM AT. The time for each iteration was recorded in hundredths of seconds. Strings were created that were exactly equal, totally different, matching halfway at the beginning, matching halfway at the end, or matching halfway in the middle. The results of these tests are reported in Figure 1, page 50, as the number of comparisons performed per second. As you can see in the figure for exactly equal strings, the procedure found virtually no change as the strings became longer. For this case, the pattern-matching procedure acted as an ordinary string-comparison function and performed approximately 8,000 comparisons per second. Totally different strings act as predicted, showing a quadratic curve in the form of N^2 (from 8,000 comparisons/second for one character to 200 comparisons/second for two 20-character strings).

PATTERN MATCHING

Strings that match at the beginning can exit quickly and those that match in the middle divide their search problem in half. (455 comparisons/second for 20-character strings.) Strings that match at the end model those that are totally unalike since nearly the entire strings are searched before the matching substring is located. (270 comparisons/second.)

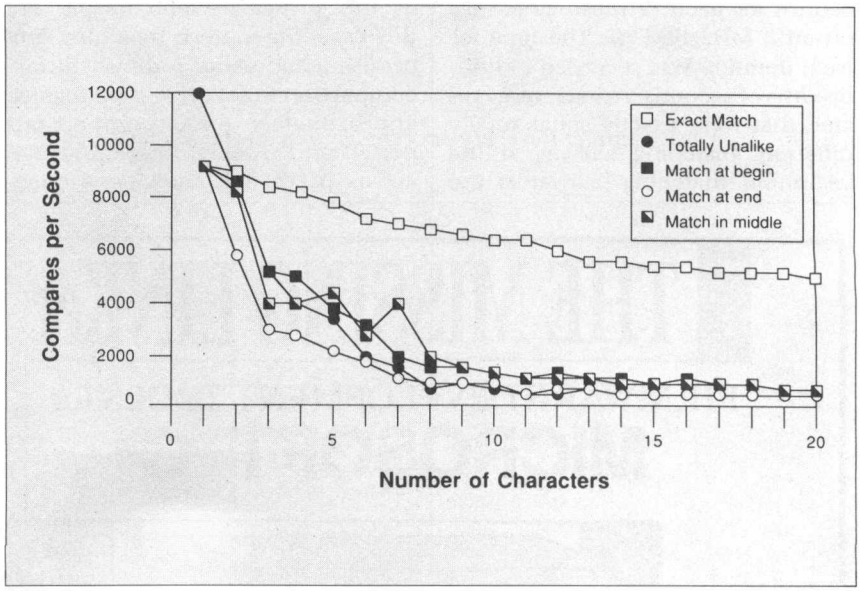


Figure 1: *Effects of timing during different compare types*

Next, two 12-character strings were analyzed for every percentage that the procedure could return. At each percentage, a wide variety of combinations of substring matches were tried. These varieties included matches at the beginning, in the middle, and at the end, and those spread differently throughout the two strings. Figure 2, page 51, displays the average time for these tests at each percentage.

Final Thoughts

Implementing this algorithm in your application can dramatically improve your software, but it does require some judgment based on the environment. The first step in interpreting the ambiguous data should be by compiling a list of the most likely alternatives. Your program's action after this should be based on both how strong and how closely grouped the candidates are. For ex-

ample, if the best match found is only a 50-percent match but all the other candidates are under 20-percent, the 50-percent match is quite likely the users' original intent. However, if there are six 90-percent matches or better, it would be best to provide the user with these matches in order of similarity, along with a convenient and rapid method of confirmation, such as a pop-up menu.

We hope this article has sparked some interest in the programming community, and we'd appreciate hearing about your applications of the algorithm. Adding pattern recognition to software has tremendous potential for improving all our lives—programmers and users alike. We might finally make "user-friendly" something more than a marketing cliche.

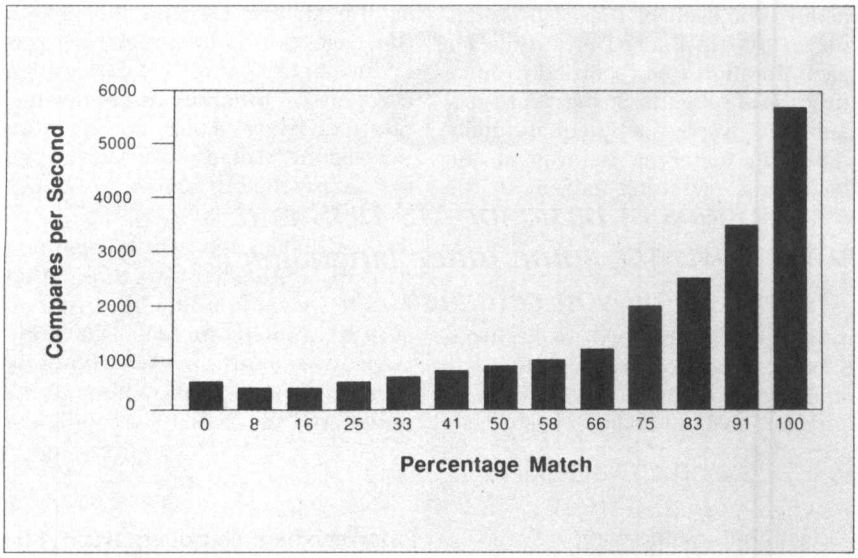


Figure 2: *Average timing for every percentage of a 12- to 12-character string match*

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to Dr. Dobb's Journal, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 68.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

GETTING DOWN TO BASICS

With the new versions of Basic for MS-DOS and OS/2, if you left Basic for some other language, maybe it's time you returned.

by Bruce Tonkin

If C is a scalpel, Basic is a Swiss Army knife. While some languages are more specialized—C for systems programming, APL for tables and arrays—Basic is more general. Rather than asking "Why Basic?", it might be better to ask "When Basic?"

Admittedly, no computer language is a universal tool. Like any other language, Basic has its own fundamental strengths and weaknesses. Also, each vendor's implementation of the language has its own particular strengths and weaknesses.

So what is it that makes Basic so basic? For one thing, it's familiar and readable. Nearly anyone can read a Basic program (though poorly structured code can make you wish you hadn't). Except for Applesoft (which lacks even an *ELSE*), spaghetti code is always the programmer's fault. Basic programs are often written by nonprogrammers precisely because Basic is so approachable. Pascal and C are harder to learn and less forgiving.

Bruce Tonkin develops and sells software for TRS-80 and MS-DOS/PC-DOS computers, he writes mostly in Basic. However, he also writes some software in C and the assembly language. You may reach him at T.N.T. Software Inc., 34069 Hainesville Rd., Round Lake, IL 60073.

While I don't claim to speak for everybody, I can't easily read Pascal programs. They're counter-intuitive: all the procedures come first, then the main logic. I don't think that way, and I suspect most people don't. Such ordering can be overridden, but it's another example of a language written for the compiler-writer instead of for the user.

The pointer indirection in C is powerful, but ultimately even more confusing than spaghetti code full of *GOTO*s and *GOSUB*s. Of the other common languages for micros, Fortran, APL, and Forth are generally hard (or downright impossible) to read without significant study. None of these supports the range of built-in features Basic does. Basic is widely known, so it's easier to find a Basic programmer and it generally costs less to hire one—an advantage for users. Since Basic has so many things built-in, Basic programmers can be more productive, and that's an advantage for the programmer.

Basic (throughout, I refer to MS-DOS GW Basic) is good at string operations. C and Pascal handle strings with user-written or library functions and have no intrinsic string type. Basic's string garbage collection has drawn fire, but dynamic strings in other languages are nontrivial and slower than compiled Basic. Many applications require dynamic strings for efficient use of memory.

Basic has excellent ways of addressing the screen, drawing shapes, and using colors. Basic supports the communications ports, can easily append data to files, and has device independence for most input and output operations. Some versions of C and Pascal have such extensions, but many must be purchased as add-ons or must be written by the user. As a developer, I find that requirement annoying, non-portable, time-wasting, and expensive. I quit using C because I saw no way I could write better or faster routines than Microsoft gave me in Basic. I have no desire to reinvent the wheel.

Another of Basic's strengths is that it's a de facto standard rather than a standard de jure. The ANSI standard for Basic is widely ignored, since ANSI Basic foolishly resembles only itself. The real standard has been, and still is, Microsoft GW Basic, ANSI notwithstanding.

Critics have often dismissed Basic as "good only for quick and dirty jobs." Those critics remember 1976-style interpreted Basic (usually Applesoft). Basic has changed a lot since 1976, and Applesoft was so bad it probably moved more programmers to Pascal than did

Borland.

Previous Basics did have problems: mandatory line numbers and no line labels; single-line functions; a clumsy interface to other languages; limited data types; and no multiline conditional statements. Those disadvantages made it difficult to write modular code, and often made debugging a nightmare.

Newer Basics make it easier to write structured code. Nearly every former disadvantage has been overcome. Still, these new versions have retained the built-in features that have made Basic so appealing.

The New Basics

The products compared here include Microsoft's MB 6.0, Quick Basics 3.0 and 4.0, and Borland's Turbo Basic 1.1. Though it's no longer being shipped by Microsoft, I decided to include Quick Basic 3.0 in this comparison for three reasons: first, because it and Turbo Basic came out at about the same time and the differences between Quick Basic and Turbo Basic are interesting; second, because Quick Basic is in some ways still the fastest Basic compiler for floating-point math speed; and third, because the environment for Quick Basic is much different than the later Microsoft products.

Any computer language is good only to the extent that it is useful. Therefore, I'll first rate these Basics on their general utility: how easily and how well they lend themselves to general-purpose business programming. Following that, I'll provide a comparative rating based upon the following criteria: speed, size of generated code, reliability, interface to other languages, portability, and environment.

In this analysis, I'll assume that any acceptable Basic will include capabilities, commands, and functions of GW Basic as a proper subset. Keep in mind the Swiss Army knife analogy. Every developer and every user is unique, and it's important that Basic serves all users well. Some of these Basics do a better job of that than others.

The competition between Borland, Microsoft, and others is leading to steadily more capable Basics. Standardized languages cannot evolve as fast. Users may demand dynamic strings and random files for Pascal, but I can't imagine ANSI adding them.

Quick Basic 3.0

Quick Basic (QB) 3.0 goes beyond GW Basic in a number of areas. The file and device I/O is essentially the same, but the maximum string length of 32,767 bytes makes it easier to read, write, and manipulate large chunks of data.

QB 3.0 supports the 8087 numeric coprocessor. For calculation-intensive programs, that's an enormous advantage. Most machines are not equipped with an 8087, though, and that creates problems. Programs can be compiled to use the 8087 and emulate it otherwise, but such programs run more slowly on machines without an 8087. The reverse is also true. Programs compiled without the emulation run well on machines without an 8087, but achieve no increased speed on machines with one.

Worse, the 8087 and non-8087 versions of these programs use a different format when storing numbers to disk. The IEEE format cannot be used by old-style programs, but either kind of program can use or write old-style floating-point numbers. So, most programmers avoid the 8087 library and stay with the old-style numbers, especially if a significant amount of data is in the old format.

Though individual numeric arrays may not exceed 64K in size, the programmer can use all available memory for numeric data. This can be an enormous advantage. Older Basics required that all such data share a single 64K segment.

QB 3.0 does not require line numbers, but error routines can identify locations only by reference to the last line number executed. This may force programmers to use unclear numbered references.

For control structures, QB 3.0 adds a *DO* loop with *WHILE* and *UNTIL* terminators, an *EXIT* statement for *FOR* and *DO* loops, and *SELECT CASE*. Block *IF* conditionals may include an *ELSEIF*.

A *CONST* (constant) declaration makes code easier to maintain and alter. Unfortunately, that declaration does not work like the C *#define* directive: *CONST* definitions do not apply outside the main module and may not include anything but the definition of a single variable.

QB 3.0 supports procedure-like subprograms, which can be separately compiled and collected into libraries. The subprograms may not be recursive, however, and the compiler does no argument type-checking or conversions. Both of these factors cause subtle and hard-to-find bugs.

QB 3.0 does not allow useful changes to the stack size or the memory used. A *CLEAR* statement can adjust the stack, but using it erases all variables (including any passed in common). That's frustrating, since the default stack size is only 768 bytes. QB 3.0 programs always grab all available system memory.

DOS calls are supported with the *CALL* statement. In practice, that means DOS services will require a short assembler routine. Assembler is not difficult to use, and the QB manual includes several examples.

For utility, QB 3.0 far surpasses GW Basic. The code can be more modular, and the language enhancements are well-suited to business programming.

Turbo Basic 1.1

Turbo Basic (TB) is much like QB 3.0, but adds interesting and valuable capabilities. As with QB 3.0, the maximum string length is 32,767 bytes. TB also adds binary file I/O. That is a powerful enhancement, and especially useful for reading "foreign" data files, such as those created by dBase. To complement this binary mode, TB adds a *SEEK* function (as in C) to position the file pointer or determine location.

The example of binary file I/O shown in the TB manual is in classic backward Pascal style. Subprograms and functions are defined first, and only the last few lines of the program use *SEEK*. Programmers comfortable with that style can use it in TB (or in QB 3.0).

TB also adds a useful long integer data type. Long integers are clearly useful for binary files, but are also good for exact dollars-and-cents arith-

(continued from page 53)

metic with an implied decimal point. Such arithmetic should be far faster than BCD math and equally accurate (though with a smaller range).

Another major difference between QB and TB is in numeric processing. TB will always use an 8087 if present and emulate it otherwise. That simplifies things somewhat, but makes for problems.

TB is slower at floating-point math than QB 3.0 programs compiled without 8087 support. Since few MS-DOS computers have an 8087, TB will perform floating-point math much more slowly than QB 3.0 can for most users.

TB will allow the programmer to use all available memory with arrays of up to 64K each. This is identical to QB 3.0.

TB handles line numbers and errors like QB 3.0, with the same disadvantages. TB does have one advantage in event-trapping, though, which can be turned on or off for selected ranges of program lines. TB programs can gain a big speed advantage from this.

TB also allows the programmer to turn off range-checking. This can add speed, but out-of-bounds references can cause fatal crashes.

Control structures are the same as in QB 3.0, with *DO* and *FOR* loops (and *EXIT* commands to leave either loop prematurely), *SELECT CASE*, and multiline conditionals including *ELSEIF*. Here, TB differs from QB 3.0. TB will not permit anything on the same line as the *ELSE* in a multiline *IF*, and QB 3.0 will. I find the QB 3.0 syntax more readable, especially for short statements. TB surpasses QB 3.0 in allowing *EXIT* for *WHILE*, *SELECT*, and block *IF*. TB also supports conditional compilation.

Many older *Basics* have implemented an increment or decrement operator. TB uses *INC* and *DEC* for this—*INC A* is less verbose only if the variable name is longer than one character.

TB allows constant declarations, but only for integer variables. That is less useful than QB 3.0.

TB allows recursive subprograms, but since TB does not support libraries or separate compilation, subprograms are less useful. As with QB 3.0, TB does no argument type-checking or automatic conversions on calls to subprograms.

TB will allow the programmer to set the amount of memory the program will use and the size of the stack (which is vital for recursion).

DOS services are easy to use in TB by using *CALL INTERRUPT*. Though using assembler routines is different than QB 3.0, it is not difficult. The *CALL INTERRUPT* statement makes the use of assembler less necessary, as well.

Static strings are nothing more than implicit arrays of type character.

TB allows COM files to be included directly, or as hex codes (inline). The inline approach enables the programmer to make small changes without re-assembly. For the kind of short routines often used in Basic, this makes a great deal of sense.

Still, Turbo Basic's lack of a link step means that it is more difficult to use code libraries than with QB 3.0. That is often a disadvantage.

For utility, TB is roughly equivalent to QB 3.0, with enough strengths to make it preferable for some operations. In particular, binary file I/O and long integer arithmetic are substantial advantages. The *CALL INTERRUPT* is another strength. Recursion is not real important. The main disadvantages of TB are the floating-point math speed and the lack of linkable libraries.

Quick Basic 4.0

QB 4.0 is not QB 3.0 with the bugs fixed. Rather, QB 4.0 is a completely new Basic that is fundamentally different from previous versions. However, QB 3.0 programs will run under QB 4.0. Only the assembler routines must be altered.

Microsoft apparently responded to TB by taking the most popular features and including them in QB 4.0 for binary file I/O, long integers, and *SEEK*. The QB 4.0 syntax is slightly different for binary file I/O. TB uses *GET$*, and QB 4.0 uses *GET*. I prefer the Microsoft syntax.

Microsoft also dropped support for non-IEEE math. Like Borland, Microsoft added functions to allow QB 4.0 to use or to write old-style floating-point numbers. Microsoft's function names are different (and more difficult to remember), but otherwise work identically.

The slower IEEE math is a disadvantage for QB 4.0 as compared to QB 3.0. Some operations take more than 30 times as long to run. If math chips were cheap or present on most machines, I could accept this more easily. As it is, many applications may never be converted from QB 3.0 to QB 4.0.

QB 4.0 allows recursive subprograms, but the stack problem remains from QB 3.0: setting the size with *CLEAR* erases all existing variables. This means recursion cannot be used for many *CHAIN*ed programs.

The TB *CALL INTERRUPT* command has also been added, with minor differences in how the registers are specified.

QB 4.0 goes beyond TB and QB 3.0 in three main areas. First, arrays are not limited to 64K in size (except dynamic strings). Second and third, user-definable data types and a new "static string" data type have been added. The first enhancement is self-explanatory, but the others require discussion.

Static strings are nothing more than implicit arrays of type character. Static strings are really not strings, though they can be treated as such in programs. Such character arrays can be placed outside of normal string space, which makes them quite useful.

Static strings are converted into dynamic strings whenever they are modified, and then reconverted back. That's slow. Even operations that do not need to create dynamic strings (for example, assignment or *MID$*

(continued from page 54)

replacement operations) will do so. This is disappointing, since a simple *MOVSB* instruction should perform the necessary operation quickly.

Static strings have another problem: their size must be declared at compile-time. If you attempt to declare a static string of length I%, where I% is determined when the program is run, you will get errors. Most often, QB 4.0 will tell you "invalid constant" and highlight the array name (not the offending constant) on the line where you attempted to *DIM* the static string or array. This forces sort and database utility programs to be hard-coded for each use.

User-defined types are similar to C structures and Pascal records. The programmer may use such a variable type to refer to any collection of fundamental data types except arrays. Many reviewers have been seduced by this similarity to Pascal records, and with reason—the Microsoft manual emphasizes it.

In practice, user-defined types are ill-suited to file I/O because such types may not include arrays. Consider a common application where a record contains 100 fields, half of which are repetitive floating-point numbers or integers (rates, prices, instrument readings, or similar quantities).

To modify those numbers within a user-defined type (set them all to zero, or add 10 to them, or discount them by 10 percent), QB 4.0 requires that each one be specifically named—loops are not possible. If the record were *FIELD*ed in the old style, the update could be performed in a loop with a significant saving in program size.

Or suppose that one file must be copied to another with more, fewer, or re-ordered fields. With user-defined types, each element must be individually specified and copied by name. This is verbose. Worse, you have no easy way to write a program capable of selecting and transferring arbitrary fields at run-time.

The no-included-array restriction makes user-defined types nearly useless for managing blocks of record pointers and other more-advanced

data structures, as well.

You may have an array of a user-defined type. There is a subtle bug that affects both these arrays and arrays of static strings. If an array of either kind is defined, that array cannot be larger than 128K in size, if the length of each element is not a power of two. This is true no matter how much memory is available. So, a program may possibly be unable to create an array of 1,025 127-byte strings, but successfully create an array of 3,000 128-byte strings.

Though they are not yet as useful as they might be, user-defined types are a valuable enhancement to Basic. Static strings can be quite useful, too. Single arrays larger than 64K are another obvious advantage. If not for the math problems, QB 4.0 would be clearly preferable to QB 3.0.

Microsoft MB 6.0

Microsoft Basic (MB) 6.0 is similar to QB 4.0, as it ought to be. An updated version of QB 4.0 is included in the package. MB 6.0 includes a set of subroutines that can make it possible to eliminate at least some unnecessary run-time support. That can make applications programs appreciably (up to about 20 percent) smaller than QB 4.0. More important, MB 6.0 will allow the programmer to use an alternate math library, regaining nearly all the speed lost in QB 4.0.

Besides QB 4.0, MB 6.0 also includes the CodeView debugger (the user-configurable Microsoft editor) a utility that can be used to create special versions of the various libraries, and a set of tools for creating OS/2 applications.

Both MB 6.0 and Quick Basic 4.0 use something that Microsoft calls "threaded P-code technology" and the company describes as "new." It is new—for Basic—but has been used for a long time for Forth and for some Pascal compiler.

With P-code, you can make small changes to a program easier and then resume execution, examine the values of variables or expressions, or even alter variables while debugging. Since only a small part of the program is changed, only a small part of the P-code is changed: Compilation (to P-code) is nearly instantaneous. In effect, debugging can proceed much as it might have for an interpreted program, but the P-code program can perform many operations nearly as fast as when truly compiled. No Basic interpreter was ever as powerful as QB 4.0, and none ever had the debugging facilities. The P-code is a substantial advantage to overall development.

P-code has an obvious advantage for Microsoft—it's highly portable. Much of QB 4.0's environment could be moved to another processor without change. Perhaps Microsoft will release QB for the 68000 or other processors. Even if something like that is not being planned, at least it is possible. With older compilers, such a product would be much more work. The only portion of the compiler that would need substantial change would be the actual code generation for executable files.

For utility, MB 6.0 is clearly better than QB 3.0, except in circumstances where math speed is absolutely essential and a math coprocessor chip will not be present. Even then, the penalty will depend to a great ex-

tent on the operations.

Ratings

Now I'll rate the different versions of Basic on criteria suited to any computer language. The first rating will be for speed.

Speed

For programmers, a compiler must perform its job quickly. If a program can be compiled rapidly, more time can be devoted to finding and fixing subtle bugs. Further, a fast compiler will make optimizing programs easier.

For users, speed simply means "how long does it take to _____." Users pay programmers, which makes the users' impressions of speed more important.

Overall, the compilation speed was excellent (less than five seconds) for all compilers tested. QB 4.0 and MB 6.0 are much slower at generating an EXE file. Each invokes BC.EXE and LINK.EXE from within the QB environment. TB is the fastest at generating an EXE file. MB 6.0 and QB 4.0 can "pseudo-compile" a program in memory faster than any other compiler and have the best facilities for tracing and debugging.

I evaluated each of the compilers for speed on 27 criteria (some compilers underwent 11 more tests). Times for each tested operation were derived by subtracting times for an appropriate empty loop. Times for a simple assignment were also subtracted for operations that included one. The times remaining are then (as nearly as possible) those for the tested operation alone.

The results are shown in Table 1, this page. Generally, TB is the fastest at integer assignments and comparisons and at printing to the screen. QB 3.0 is fastest at nearly everything else, though MB 6.0 is just as fast (or slower by only trifling amounts) on more than half the tests. QB 4.0 and TB are slow at most floating-point operations, taking more than 20 times as long as QB 3.0 for most assignments and comparisons. TB is much slower than QB 4.0 or MB 6.0 at long-integer operations, taking as much as 160 times as long as MB 6.0 to perform a long integer add.

These tests are incomplete. A definitive set would require a far larger article. TB is far better at string manipulation than the tests show. With string arrays filling most available memory, TB actually gets faster at string operations, while the Microsoft product get exponentially slower. By filling string space appropriately, you can make TB faster at string operations than any QB by any factor desired.

Size

File sizes are shown in Table 2, this page. Generally, TB and QB 3.0 produced EXE files of comparable size.

Time, in Seconds, per 1,000,000 Operations

	QB 3.0	QB 4.0	BASIC 6.0	TB
Empty integer loop	1.21	1.21	1.21	2.09
Integer assignment	1.15	1.13	1.13	1.04
Integer add	.30	.49	.50	.44
Integer subtract	.33	.46	.49	.44
Integer multiply	1.92	2.45	2.27	2.75
Integer divide	3.36	3.43	3.44	4.01
String assignment	77.09	78.50	79.04	176.91
String MID$ operation	20.78	24.61	25.20	146.18
String concatenation	1657.81	954.84	1661.15	1848.25
Single-precision assignment	5.71	201.23	20.40	269.46
Single-precision add	23.97	128.83	24.41	206.25
Single-precision subtract	24.80	129.36	25.87	206.52
Single-precision multiply	34.30	-16.19	35.50	176.62
Single-precision divide	35.98	7.48	47.68	207.92
Error, 100K single-p mult/div	-1.06E-05	-1.19E-07	-1.19E-07	-1.19E-07
Single-precision exponential	766.80	3061.84	1296.36	2805.25
Double-precision assignment	6.26	211.41	23.06	279.07
Double-precision add	45.75	143.28	49.99	152.75
Double-precision subtract	47.19	143.87	51.52	151.10
Double-precision multiply	83.13	199.49	91.66	185.67
Double-precision divide	84.22	226.45	121.82	212.81
Error, 100K double-p mult/div	-4.82E-13	-2.22E-16	2.22E-16	-6.66E-16
Double-precision exponential	2491.80	6377.23	3046.33	2851.74
Integer comparison	2.58	3.20	2.58	2.31
Single-precision comparison	19.72	402.00	42.68	444.48
Double-precision comparison	20.37	412.89	46.30	452.70
Conditional int assignment	4.51	4.77	4.67	2.88
Conditional assignment*	4.72	4.88	4.34	3.37
Print 1K 70-byte strings**	26.47	10.00	10.05	9.88
Empty long integer loop	——	14.45	14.28	1071.42
Long integer assignment	——	1.62	1.71	274.60
Long integer add	——	15.47	15.26	1236.16
Long integer subtract	——	15.41	15.21	1235.68
Long integer multiply	——	26.11	25.82	1252.45
Long integer divide	——	33.76	33.66	1521.17
Fixed string assignment	——	52.38	51.92	———
Fixed string MID$ operation	——	23.44	24.12	———
Fixed string concatenation	——	24.61	24.78	———
Long integer comparison	——	10.81	10.98	449.70
Print 1K 70-byte static strings**	——	10.16	10.10	———

*Reversed order. Unreversed should be faster.
**To the screen. Monochrome display, MDA card.
Note: the negative and small times for single-precision multiply and divide for QB 4.0 don't mean those operations took place in negative time, but that they took less time to execute than a simple assignment. Perhaps there are unnecessary checking or conversion operations going on during the simple assignment. Considering the big slow-down from QB 3.0, that seems likely.

Table 1: *Benchmark operations per million*

File Sizes

File	QB 3.0	QB 4.0	BASIC 6.0	Turbo BASIC
BENCHOLD.BAS	7,400	7,527	7,417	7,460
BENCHOLD.OBJ	12,522	12,092	14,178	———
BENCHOLD.EXE	45,584	37,569	35,407	44,691
BENCHNEW.BAS	———	2,887	2,887	1,788
BENCHNEW.OBJ	———	6,393	7,035	———
BENCHNEW.EXE	———	34,073	29,029	38,784

Note: Benchold.bas was slightly modified for QB 4.0 and TB because of their very slow performance on a number of the benchmark tests. Benchnew.bas was modified for TB, since TB does not have fixed-length strings. All programs were saved as text files and compiled with all trapping and checking options turned off (where that was possible).

Table 2: *File sizes*

(continued from page 60)

QB 4.0 produced significantly smaller EXE files than either, and MB 6.0 produced the smallest.

None of these compilers seems to do much optimization, and all include too much "dead" code in the final EXE. As an educated guess, BENCHOLD.EXE should have been smaller than 10K, and BENCHNEW.EXE less than 4K. All these compilers included at least 25K of unneeded code.

Reliability

Each of the compilers was used for some months or years. Generally, all are reasonably "clean." Basic 4.0 is the least reliable, but that should be expected since it is the newest. MB-6.0 is an updated version of QB 4.0, and has fewer bugs.

In my experience, the Microsoft products make too many assumptions about the underlying hardware and DOS. MB 6.0 is no different. A program I compiled for the OS/2 environment failed with an "out of memory" error on a PS/2 Model 50 with 2 Mbyte of memory; ran but had difficulties with the Norton Guide to the OS/2 API on a Compaq DeskPro 386; and ran fine on an AT clone with a standard keyboard, but failed to recognize an enhanced keyboard on the same machine.

QB 4.0 will not correctly deallocate memory if run under DOS 2.1. Programs that use dynamic arrays will not chain or run other programs. If programs compiled with QB 4.0 are run on machines with older BIOS chips, the cursor is mangled on exit. The QB 4.0 and MB 6.0 editors must be patched to work correctly on keyboards with a separate cursor control pad, and programs compiled with either QB 4.0 or MB 6.0 may have trouble with nonstandard keyboards.

The QB 4.0 editor environment has an aggravating tendency to lock up the computer or to crash. Usually, this happens when a program is large or when many breakpoints or watchpoints have been set. Most of the problems go away under DOS 3.x, but that is of little comfort if you have only DOS 2.1.

Some of these problems derive from inadequate testing, which Microsoft has promised to correct. Some come from undoubted bugs in various versions of DOS or with older versions of the BIOS. Others are more ill-considered. I see no reason for a compiler to go directly to the hardware or generate code that does. I'm willing to write routines that go directly to the screen and to take responsibility for any problems, but I'm unhappy about compilers that create hardware-dependent problems for me.

Every one of these compilers has trouble with some keyboards. The TB editor will not recognize CTRL-BREAK on one of mine. Every compiler can be forced to generate programs that work with all keyboards (by writing an assembler routine that uses normal DOS services to capture keystrokes).

If you intend to use any of these compilers, I recommend that you join a user's group or a SIG on one of the computer networks (CompuServe, GEnie, and BIX are especially good). No product is perfect, and the more complex products have more bugs. Each of these products is complex. Considering that, they are remarkably bug-free.

Interfacing to Other Languages

Microsoft's QB 4.0 and MB 6.0 win this contest. Microsoft provides explicit support for linking either product with C, Pascal, Fortran, and assembler. The manuals contain detailed examples of working programs with such interfaces.

Borland has a reasonable treatment of assembler interface, but no real provision for other languages. The lack of a link step makes such an interface difficult, at best.

Code Portability

All versions generally are portable to about the same extent: good within the MS-DOS world (but remember my earlier comments about keyboards) and poor elsewhere. All show great similarities to Macintosh Basic, which is no surprise since Microsoft wrote it. I've seen reports that another company has released a version of Basic (compatible with QB) for Unix systems. Microsoft Basic for the Xenix system has not changed much since the days of CP/M BASCOM 5.3.

If Microsoft were to release a QB product for Unix, that would presumably give many people a reason to choose Unix rather than OS/2. I suspect, for that reason, that no QB will be released for Unix or Xenix for at least several more years.

Programming Environment

This part is tricky. What I like, you may not. I will try to separate (or at least note) my personal preferences.

Microsoft has shown a steadily increasing trend to a windowed environment with dialog boxes and an equally obvious preference for a mouse. I find these trends distressing and counter-productive. I know three other programmers who use QB 4.0 on a frequent basis. I am the only one who even uses the editor, and I don't like it. Considering the powerful debugging commands in QB 4.0, that's a strong indictment of the editor.

Few programmers seem to like dialog boxes. That may be because we are programmers and not first-time computer users. Further, all programmers have a style. Some like to capitalize everything, and some prefer all lowercase. Some like to separate variables from operators, others do not. The Microsoft QB 4.0 editor will not permit many variations. It will capitalize all keywords, separate all variables and operators, and refuse to permit a line-continuation underline character. Of all the bad things the Microsoft editor does, mandatory formatting is the worst. I want source code to remain as I typed it!

TB's editor is nicer than any of Microsoft's. It is more like WordStar, which means it demands less effort to learn and use. Further, it supports the block read and write commands, and none of Microsoft's editors do (a significant problem). The TB environment is also window-oriented, but less obtrusively so while editing. Further, the commands and prompts appear in pre-

dictable locations (not true for the QB 4.0 and MB 6.0 dialog boxes).

The results obtained in the editor environment should match those obtained from the EXE file. QB 4.0 has some differences in math, which have caused problems with my double-precision routines.

If you need debugging help, any of the Microsoft products are superior to TB. While TB will allow you to trace execution of your program and step by line number or label, the Microsoft products allow you to single-step by statement, track values of variables or expressions, and compute expressions (even open or close files). To set a breakpoint with TB or QB 3.0, you must insert a command in your source code. QB 4.0 and MB 6.0 allow breakpoints to be set with a single keypress and no alteration of the source.

QB 4.0 and MB 6.0 are better at detecting errors while entering source code. Missing parentheses and most illegal commands (or statements) are detected instantly and the error location is shown. None of these compilers is good at pinpointing errors such as a missing *END IF*, *WEND*, or *NEXT* (and it is difficult to see how they could be). The messages and locations given for such errors are generally misleading or just plain wrong.

Run-time errors are easier to find and fix with TB, since the editor can point to the location given by the run-time support package. For the QB products, finding the location of a run-time error generally means recompiling and generating a MAP file, then manually searching the (large) resulting text file.

Summary and Conclusions

Overall, these versions of Basic are greatly improved over those of just a few years ago. The improvements have been especially strong in documentation, modularity, and overall power. Despite that, these Basics lack support for some useful data types: unsigned integers, pointers, and BCD numbers. Matrix operations, a built-in sort command, and ISAM support are still lacking. Code optimization is weak or nonexistent, but overall performance is acceptable because of the great number of built-in functions.

Any of these newer Basics are acceptable products for large projects. For programs that use mainly integer math, must manipulate large string arrays, or are limited by screen display speed, TB is a good choice. QB 4.0 has better long integer and floating-point speed than TB, with very nearly the same screen speed. QB 3.0 is fastest but least accurate for math, slowest at screen display, and has no support for long integers. MB 6.0 is second-fastest at overall math speed, with greatest math accuracy and almost the same display speed as TB, but it is the most expensive product.

Programs that need single arrays of larger than 64K, user-definable data types, OS/2 support, or static strings must use either QB 4.0 or MB 6.0. Programs that must run under DOS 2.1 should be compiled with QB 3.0 or TB. Recursion is not supported by QB 3.0.

Because of these requirements and limitations, the professional programmer should have at least TB and one of the QB products. I'd recommend MB 6.0 and TB if you want only two compilers, but QB 3.0 and 4.0 are another attractive pair. For general utility, a good Basic compiler is hard to beat—and any of these products is better than merely good!

Availability

All source code for this article in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 76.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

HYPERANIMATION

Listing One *(text begins on page 18.)*

```
1988, Bright Star Technology, Inc.
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--*********************--
--*   Stack functions       *--
--*********************--

on startup

  --*********************************************************--
  --*  Open up the driver using ELON as our synthetic actor and move      *--
  --*  him to where we would like to see him on the screen.                *--
  --*********************************************************--
   RAVE "|ACTOR ELON|"
   RAVE "|MOVE TOP 100 LEFT 150|"

end startup

-------------------------------------------------------------------------

function scroll_line how_many_lines

  --*********************************************************--
  --*   This function will quickly scroll the field "prose line" the          *--
  --*   number of lines passed in the parameter how_many_lines.          *--
  --*********************************************************--
  repeat how_many_lines
    set the scroll of field "prose line"B
    to the scroll of field "prose line"B
    + textHeight of field "prose line"
  end repeat

end scroll_line

-------------------------------------------------------------------------

function show_and_tell this_text

  --*********************************************************--
  --*  This function shows what the actor is saying in the field            *--
  --*  face line (located underneath the actor's face) and then says it.   *--
  --*********************************************************--
  put this_text into card field "faceline"
  RAVE card field "faceline"
  put empty into card field "faceline"

end show_and_tell



-------------------------------------------------------------------------
-------------------------------------------------------------------------


--*********************--
 --*   Button "A limerick"   *--
--*********************--
on mouseUp

  --*********************************************************--
  --*  When this button is pressed, the limerick field is reset to show  *--
  --*  the first line of the limerick on the screen.                       *--
  --*********************************************************--
  set the scroll of field "prose line" to 60

  --*********************************************************--
  --*  Each line of the limerick is read from the field  "prose line",      *--
  --*  pronounced, and then scrolled upwards by calling the function     *--
  --*  scroll_line.                                                        *--
  --*********************************************************--
  repeat with prose_count = 7 to 11
     RAVE line prose_count of card field "prose line"
     put scroll_line(1) into nothing
  end repeat

  --*********************************************************--
  --*  Finally we pause a moment, before scrolling the limerick up off   *--
  --*  the screen.                                                         *--
  --*********************************************************--
  wait for 3 seconds
  put scroll_line(7) into nothing

end mouseUp


-------------------------------------------------------------------------
-------------------------------------------------------------------------

--*********************--
--*   Button "Bad Rating"   *--
--*********************--

 on mouseUp

   put show_and_tell ("Sure, let's hear you try it!") into nothing

   set the scroll of field "prose line" to 60
   put show_and_tell ("Go ahead, I am listening ..." ) into nothing

   put scroll_line(7) into nothing

   put show_and_tell ("What's Wrong?" ) into nothing
   put show_and_tell ("Cat got your tongue?" ) into nothing

 end mouseUp

 -------------------------------------------------------------------------

 --*********************--
 --*   Button "Good Rating" *--
  --*********************--

 on mouseUp

   put show_and_tell ("Thank You!") into nothing
   put show_and_tell ("You are too kind!" ) into nothing

 end mouseUp
```

End Listing

PATTERN MATCHING

Listing One *(Text begins on page 46.)*

```
        TITLE   SIMIL.ASM written by John W. Ratcliff and David E. Metzener

; November 10, 1987
; Uses the Ratcliff/Obershelp pattern recognition algorithm.
; This program provides a new function to C on an 8086 based machine.
; The function SIMIL returns a percentage value corresponding to how
; alike any two strings are.  Be certain to upper case the two strings
; passed if you are not concerned about case sensitivity.
; NOTE:  This routine is for SMALL model only.  As an exercise for
; the student, feel free to convert it to LARGE.

_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP



_DATA   SEGMENT

ststr1l dw   25 dup(?)              ; contains lefts for string 1
ststr1r dw   25 dup(?)              ; contains rights for string 1
ststr2l dw   25 dup(?)              ; contains lefts for string 2
ststr2r dw   25 dup(?)              ; contains rights for string 2
stcknum dw   ?                      ; number of elements on the stack
score   dw   ?                      ; the #of chars in common times 2
total   dw   ?                      ; total #of chars in string 1 and 2
cl1     dw   ?                      ; left of string 1 found in common
cr1     dw   ?                      ; right of string 1 found in common
cl2     dw   ?                      ; left of string 2 found in common
cr2     dw   ?                      ; right of string 2 found in common
s2ed    dw   ?                      ; the end of string 2 used in compare

_DATA   ENDS

        public _simil


_TEXT   SEGMENT

_simil  proc   near
; This routine expects pointers passed to two character strings, null
; terminated, that you wish compared.  It returns a percentage value
; from 0 to 100% corresponding to how alike the two strings are.
;                       +4      +6
; usage: simil(char *str1,char *str2)
; The similarity routine is composed of three major components
; pushst    ---- pushes a strings section to be compared on the stack
; popst     ---- pops a string section to be examined off of the stack
; compare   ---- finds the largest group of characters in common between
;                any two string sections
; The similarity routine begins by computing the total length of both
; strings passed and placing that value in TOTAL.  It then takes
; the beginning and ending of both strings passed and pushes them on
; the stack.  It then falls into the main line code.
; The original two strings are immediately popped off of the stack and
; are passed to the compare routine.  The compare routine will find the
; largest group of characters in common between the two strings.
; The number of characters in common is multiplied times two and added
; to the total score.  If there were no characters in common then there
; is nothing to push onto the stack.  If there are exactly one character
; to the left in both strings then we needn't push it on the stack.
; (We already know they aren't equal from the previous call to compare.)
; Otherwise the characters to the left are pushed onto the stack.  These
; same rules apply to characters to the right of the substring found in
; common.  This process of pulling substrings off of the stack, comparing
; them, and pushing remaining sections on the stack is continued until
; the stack is empty.  On return the total score is divided by the
; number of characters in both strings.  This is multiplied time 100 to
; yield a percentage.  This percentage similarity is returned to the
; calling procedure.

        push    bp              ;save BP reg.
        mov     bp,sp           ;save SP reg in BP for use in program
        push    es              ;save the ES segment register
        mov     ax,ds           ;copy DS segement register to ES
        mov     es,ax
        xor     ax,ax           ;zero out AX for clearing of SCORE var.
        mov     score,ax        ;zero out SCORE
        mov     stcknum,ax      ;initalize number of stack entries to 0
        mov     si,[bp+4]       ;move beginning pointer of string 1 to SI
        mov     di,[bp+6]       ;move beginning pointer of string 2 to DI
        cmp     [si],al         ;is it a null string?
        je      strerr          ;can't process null strings.
        cmp     [di],al         ;is it a null string?
        jne     docmp           ;neither is a null string so process them
strerr: jmp     donit           ;exit routine
docmp:  push    di              ;save DI because of SCAS opcode
        push    si              ;save SI because of SCAS opcode
        xor     al,al           ;clear out AL to search for end of string
        cld                     ;set direction flag to forward
        mov     cx,-1           ;make sure we repeat the correct # of times
        repnz   scasb           ;scan for string delimiter in string 2
        dec     di              ;point DI to '$00' byte of string 2
        dec     di              ;point DI to last character of string 2
        mov     bp,di           ;move DI to BP where it is supposed to be
        pop     di              ;restore SI into DI for SCAS (string 1)
        repnz   scasb           ;scan for string delimiter in string 1
        not     cx              ;do one's compliment for correct length of st
        sub     cx,2            ;subtract the two zero bytes at the end of st
        mov     total,cx        ;store string 2's length
        dec     di              ;point DI to '$00' byte of string 1
        dec     di              ;point DI to last character of string 1
        mov     bx,di           ;move DI to BX where it is supposed to be
        pop     di              ;restore DI to what it should be
        call    pushst          ;Push values for the first call to SIMILIARITY
main:   cmp     stcknum,0       ;is there anything on the stack?
        je      done            ;No, then all done!
        call    popst           ;get regs. set up for a COMPARE call
        call    compare         ;do compare for this substring set
        cmp     dx,0            ;if nothing in common then nothing to push
```

```
        je      main            ;try another set
        shl     dx,1            ;*2 for add to score
        add     score,dx        ;add into score
        mov     bp,stcknum      ;get number of entry I want to look at
        shl     bp,1            ;get AX ready to access string stacks
        mov     si,[ststr1l+bp] ;move L1 into SI or L1
        mov     bx,cl1          ;move CL1 into BX or R1
        mov     di,[ststr2l+bp] ;move L2 into DI or L2
        mov     cx,cl2          ;move CL2 into CX t emporarily
        mov     ax,[ststr1r+bp] ;get old R1 off of stack
        mov     cl1,ax          ;place in CL1 temporarily
        mov     ax,[ststr2r+bp] ;get old R2 off of stack
        mov     cl2,ax          ;save in CL2 temporarily
        mov     bp,cx           ;place CL2 into BP
        cmp     bx,si           ;compare CL1 to L1
        je      chrght          ;if zero, then nothing on left side string 1
        cmp     bp,di           ;compare CL2 to L2
        je      chrght          ;if zero, then nothing on left side string 2
        dec     bx              ;point to last part of left side string 1
        dec     bp              ;point to last part of left side string 2
        cmp     bx,si           ;only one character to examine?
        jne     pushit          ;no->we need to examine this
        cmp     bp,di           ;only one character in both?
        je      chrght          ;nothing to look at if both only one char
pushit: call    pushst          ;push left side on stack
chrght: mov     si,cr1          ;move CR1 into SI or L1
        mov     bx,cl1          ;move R1 into BX or R1
        mov     di,cr2          ;move CR2 into DI or L2
        mov     bp,cl2          ;move R2 into BP or R2
        cmp     si,bx           ;compare CR1 to R1
        je      main            ;if zero, then nothing on right side string 1
        cmp     di,bp           ;compare CR2 to R2
        je      main            ;if zero, then nothing on right side string 2
        inc     si              ;point to last part of right side string 1
        inc     di              ;point to last part of right side string 2
        cmp     bx,si           ;only one character to examine?
        jne     push2           ;no->examine it
        cmp     bp,di           ;only one character to examine in both?
        je      main            ;yes->get next string off of stack
push2:  call    pushst          ;push right side on stack
        jmp short main          ;do next level of compares
done:   mov     ax,score        ;get score into AX for MUL
        mov     cx,100          ;get 100 into CX for MUL
        mul     cx              ;Multiply by 100
        mov     cx,total        ;get total characters for divide
        div     cx              ;Divide by total
donit:  pop     es              ;Restore ES segment register to entry value
        pop     bp              ;Restore BP back to entry value
        ret                     ;Leave with AX holding % similarity
_simil  endp

compare proc    near
; The compare routine locates the largest group of characters between string 1
;   and string 2.  This routine assumes that the direction flag is clear.
; Pass to this routine:
;   BX    = R1 (right side of string 1)
;   DS:SI = L1 (left side of string 1)
;   ES:DI = L2 (left side of string 2)
;   BP    = R2 (right side of string 2)
;
; This routine returns:
;   DX    = # of characters matching
;  CL1    = Left side of first string that matches
;  CL2    = Left side of second string that matches
;  CR1    = Right side of first string that matches
;  CR2    = Right side of second string that matches
; The compare routine is composed of two loops, an inner and an outer.
; The worst case scenario is that there are absolutely no characters in
; common between string 1 and string 2.  In this case N x M compares are
; performed.  However, whan an equal condition occurs in the inner
; loop, then the next character to be examinded in string 2 (for this loop)
; is advanced by the number of characters found equal.  Whenever a new
; maximum number of characters in common is found then the ending location
; of both the inner and outer loop is backed off by the difference between
; the new max chars value and the old max chars value for both loops.  In
; short if 5 characters have been found in common part of the way through
; the search then we can cut our search short 5 characters before the
; true end of both strings since there is no chance of finding better than
; a 5 character match at that point.  This technique means that an exact
; equal match will require only a single compare and combinations of other
; matches will proceed as effciently as possible.

        mov     s2ed,bp         ;store end of string 2
        xor     dx,dx           ;Init MAXCHARS
forl3:  push    di              ;Save start of string 2
forl4:  push    di              ;Save start of string 2
        push    si              ;Save start of string 1
        mov     cx,s2ed         ;Set up for calc of length of string 1
        sub     cx,di           ;get length of string 1 -1
        inc     cx              ;make proper length
        push    cx              ;Save starting length of string 1
        repz    cmpsb           ;compare strings
        jz      equal           ;if equal, then skip fixes
        inc     cx              ;inc back because CMPS decs even if not equal
equal:  pop     ax              ;get starting length of string1
        sub     ax,cx           ;get lenght of common characters
        jnz     newmax          ;more than 0 chars matched
        pop     si              ;get back start of string 1
        pop     di              ;get back start of string 2
reent:  inc     di              ;Do the next character no matter what
reent2: cmp     di,bp           ;Are we done with string 2?
        jle     forl4           ;No, then do next string compare
        pop     di              ;get back start of string 2
        inc     si              ;next char in string 1 to scan
        cmp     si,bx           ;Are we done with string 1?
        jle     forl3           ;No, then do next string compare

        ret                     ;MAXCHARS is in DX register
; We branch downwards for both newmax and newmx2 because on the
; 8086 line of processors a branch not taken is much faster than
; one which is.  Since the not equal condition is to be
; found most often and we would like the inner loop to execute as quickly
; as possible we branch outside of this loop on the less frequent
; occurrence.  When a match and or a new maxchars is found we branch down to
; these two routines, process the new conditions and then branch back up
; to the main line code.
newmax: cmp     ax,dx           ;greater than MAXCHARS?
        jg      newmx2          ;yes, update new maxchars and pointers
        pop     si              ;get back start of string 1
        pop     di              ;get back start of string 2
        add     di,ax           ;Skip past matching chars
        jmp short reent2        ;re-enter inner loop
newmx2: pop     si              ;get back start of string 1
        pop     di              ;get back start of string 2
        mov     cl1,si          ;put begin of match of string 1
        mov     cl2,di          ;put begin of match of string 2
        mov     cx,ax           ;save new maxchars
        sub     ax,dx           ;get delta for adjustment to ends of strings
        sub     bx,ax           ;adjust end of string 1
        sub     bp,ax           ;adjust end of string 2
        mov     dx,cx           ;new maxchars
        dec     cx              ;set up for advance to last matching char
        add     di,cx           ;advance to last matching char string 2
        mov     cr2,di          ;put end of match of string 2
        add     cx,si           ;advance to last matching char string 1
        mov     cr1,cx          ;put end of match of string 1
        jmp short reent         ;re-enter inner loop
compare endp

pushst  proc near
; On entry:
;   BX    = R1 (right side of string 1)
;   DS:SI = L1 (left side of string 1)
;   ES:DI = L2 (left side of string 2)
;   BP    = R2 (right side of string 2)

        mov     cx,bp           ;save R2
        mov     bp,stcknum
        shl     bp,1            ;*2 for words
        mov     [bp+ststr1l],si ;put left side of string 1 on stack
        mov     [bp+ststr1r],bx ;put right side of string 1 on stack
        mov     [bp+ststr2l],di ;put left side of string 2 on stack
        mov     [bp+ststr2r],cx ;put right side of string 2 on stack
        inc     stcknum         ;Add one to number of stack entries
        mov     bp,cx           ;Restore R2
        ret
pushst  endp


popst   proc near
;   BX    = R1 (right side of string 1)
;   DS:SI = L1 (left side of string 1)
;   ES:DI = L2 (left side of string 2)
;   BP    = R2 (right side of string 2)

        dec     stcknum         ;point to last entry in stack
        mov     bp,stcknum      ;get number of stack entries
        shl     bp,1            ;*2 for words
        mov     si,[bp+ststr1l] ;restore left side of string 1 from stack
        mov     bx,[bp+ststr1r] ;restore right side of string 1 from stack
        mov     di,[bp+ststr2l] ;restore left side of string 2 from stack
        mov     bp,[bp+ststr2r] ;restore right side of string 2 from stack
        ret
popst   endp


_TEXT   ENDS

        END
```

End Listings

BASICS

Listing One *(Text begins on page 54)*

```
DEFLNG A-Z
DIM t!(28)
OPEN "bas6new.tim" FOR OUTPUT AS #1
DIM x1 AS STRING * 1
DIM x26 AS STRING * 26
DIM x70 AS STRING * 70
DIM x10000 AS STRING * 10000

'time for a raw long integer loop,
     executed 1,000,000 times.
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for 1,000,000 long
     integer assignments.
y = 5&: z = -5&
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 long integer adds
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x + y
  NEXT j
NEXT i
t!(2) = TIMER - t! - t!(1)

'time for 1,000,000 long integer
     subtracts
x = 0
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x - y
  NEXT j
NEXT i
t!(3) = TIMER - t! - t!(1)

'time for 1,000,000 long
     integer multiplies
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000 long
     integer divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'time for 100,000 fixed
     string assignments
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x26 = "abcdefghijklmnopqrstuvwxyz"
  x26 = "zyxwvutrsqponmlkjihgfedcba"
  NEXT j
NEXT i
t!(6) = (TIMER - t! - t!(0) / 10) / 2

'time for 100,000 fixed
     string MID$ operations
k = 17
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  MID$(x26, k, 1) = "d"
  NEXT j
NEXT i
t!(7) = TIMER - t! - t!(0) / 10

'time for 10,000 fixed
     string "concatenations"
x$ = ""
t! = TIMER
FOR i = 1 TO 10000
  MID$(x10000, i, 1) = "a"
NEXT i
t!(8) = TIMER - t! - t!(0) / 100

'following are logical
     comparisons and operators

'time for 1,000,000 long
     integer comparisons
x = 5&: y = -5&
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF i < y THEN x = 1
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)

'screen output: print 1,000
     70-byte fixed strings
x70 = STRING$(70, 66)
t! = TIMER
FOR i = 1 TO 1000
  PRINT x70
NEXT i
t!(28) = TIMER - t! - t!(0) / 1000

'print results of benchmark
PRINT #1, "Raw long integer loop
  1,000,000 iterations:"; TAB(45); t!(0)
PRINT #1, "1,000,000 long
  integer assignments:"; TAB(45); t!(1)
PRINT #1, "1,000,000 long
  integer additions:"; TAB(45); t!(2)
```

```
PRINT #1, "1,000,000 long
  integer subtractions:"; TAB(45); t!(3)
PRINT #1, "1,000,000 long
  integer multiplications:"; TAB(45); t!(4)
PRINT #1, "1,000,000 long
  integer divisions:"; TAB(45); t!(5)
PRINT #1, "100,000 fixed
  string assignments:"; TAB(45); t!(6)
PRINT #1, "100,000 fixed
  string MID$ operations:"; TAB(45); t!(7)
PRINT #1, "10,000 fixed
  string concatenations:"; TAB(45); t!(8)
PRINT #1, "1,000,000 long
  integer comparisons:"; TAB(45); t!(23)
PRINT #1, "Print 1,000 70-byte
  strings to the screen:"; TAB(45); t!(28)
END
```

End Listing One

Listing Two

```
DEFLNG A-Z
DIM t!(28)
OPEN "c:tbnew.tim" FOR OUTPUT AS #1

'time for a raw long integer loop,
      executed 1,000,000 times.
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for 1,000,000
      long integer assignments.
y = 5: z = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 long integer adds
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  x = x + y
  NEXT j
NEXT i
t!(2) = TIMER - t! - t!(1)

'time for 1,000,000
      long integer subtracts
x = 0
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  x = x - y
  NEXT j
NEXT i
t!(3) = TIMER - t! - t!(1)

'time for 1,000,000
      long integer multiplies
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  x = i * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000
      long integer divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'following are logical
      comparisons and operators

'time for 1,000,000
      long integer comparisons
x = 5: y = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
    IF i < y THEN x = 1
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)

'print results of benchmark
PRINT #1, "Raw long integer loop,
  1,000,000 iterations:"; TAB(45); t!(0)*100
PRINT #1, "1,000,000 long
  integer assignments:"; TAB(45); t!(1)*100
PRINT #1, "1,000,000 long
  integer additions:"; TAB(45); t!(2)*100
PRINT #1, "1,000,000 long
  integer subtractions:"; TAB(45); t!(3)*100
PRINT #1, "1,000,000 long
  integer multiplications:"; TAB(45); t!(4)
                                      *100
PRINT #1, "1,000,000 long
  integer divisions:"; TAB(45); t!(5)*100
PRINT #1, "1,000,000 long
  integer comparisons:"; TAB(45); t!(23)*100
END
```

End Listing Two

Listing Three

```
DEFINT A-Z
DIM t!(28)
OPEN "bas6.tim" FOR OUTPUT AS #1
'time for a raw integer loop,
      executed 1,000,000 times.
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for a integer assignment
      loop, executed 1,000,000 times.
y = 5: z = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 integer adds
y = 5: z = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x + y
  x = x + z
  NEXT j
NEXT i
t!(2) = (TIMER - t! - t!(0)) / 2 - t!(1)
'time for 1,000,000 integer subtracts
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x - y
  x = x - z
  NEXT j
NEXT i
t!(3) = (TIMER - t! - t!(0)) / 2 - t!(1)

'time for 1,000,000
      integer multiplies
k = 7
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = k * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000
      integer divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'time for 100,000
      string assignments
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x$ = "abcdefghijklmnopqrstuvwxyz"
  x$ = "zyxwvutsrqponmlkjihgfedcba"
  NEXT j
NEXT i
t!(6) = (TIMER - t! - t!(0) / 10) / 2

'time for 100,000
      string MID$ operations
x$ = "abcdefghijklmnopqrstuvwxyz"
k = 17
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  MID$(x$, k, 1) = "d"
  NEXT j
NEXT i
t!(7) = TIMER - t! - t!(0) / 10

'time for 10,000 string
      concatenations
x$ = ""
t! = TIMER
FOR i = 1 TO 10000
  x$ = x$ + "a"
NEXT i
t!(8) = TIMER - t! - t!(6) / 10 - t!(0)
                                 / 100
'time for a single-precision assignment
      loop, executed 1,000,000 times.
y! = 5!: z! = -5!
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x! = y!
  x! = z!
  NEXT j
NEXT i
t!(9) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 single-precision adds
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x! = x! + y!
    x! = x! + z!
  NEXT j
NEXT i
t!(10) = (TIMER - t! - t!(0)) / 2 - t!(9)

'time for 1,000,000 single-precision
      subtracts
x! = 0!
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x! = x! - y!
    x! = x! - z!
  NEXT j
NEXT i
t!(11) = (TIMER - t! - t!(0)) / 2 - t!(9)

'time for 100,000 single-precision
      multiplies
x! = 1!
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x! = x! * 1.00001
  NEXT j
NEXT i
t!(12) = TIMER - t! - t!(0) / 10 - t!(9) / 10

'time for 100,000 single-precision
      divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x! = x! / 1.00001
  NEXT j
NEXT i
t!(13) = TIMER - t! - t!(0) / 10 - t!(9) / 10

'error in single-precision
      multiply/divide
t!(14) = x! - 1!

'time for 10,000 single-precision
      exponentiations
x! = 100!
t! = TIMER
FOR i = 1 TO 10000
  x! = x! ^ .999999
NEXT i
t!(15) = TIMER - t! - t!(0) / 100 - t!(9) / 100

'time for a double-precision assignment
      loop, executed 1,000,000 times.
y# = 5.5#: z# = -5.5#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x# = y#
  x# = z#
  NEXT j
NEXT i
t!(16) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 double-precision adds
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x# = x# + y#
  NEXT j
NEXT i
t!(17) = TIMER - t! - t!(16) - t!(0)

'time for 1,000,000 double-precision
      subtracts
x# = 0#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x# = x# - y#
  NEXT j
NEXT i
t!(18) = TIMER - t! - t!(16) - t!(0)

'time for 100,000 double-precision
      multiplies
x# = 1#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# * 1.00001#
  NEXT j
NEXT i
t!(19) = (TIMER - t! - t!(0) / 10) - t!(16) / 10

'time for 100,000 double-precision
      divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# / 1.00001#
  NEXT j
NEXT i
t!(20) = (TIMER - t! - t!(0) / 10) - t!(16) / 10

'error in double-precision
      multiply/divide

t!(21) = x# - 1#

'time for 10,000 double-precision
      exponentiations
x# = 100#
t! = TIMER
FOR i = 1 TO 10000
  x# = x# ^ .999999#
NEXT i
t!(22) = (TIMER - t! - t!(0) / 100) - t!(16) / 100

'following are logical comparisons
      and operators

'time for 1,000,000 integer comparisons
x = 0
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF i < x THEN x = 1
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)

'time for 1,000,000 single-precision
      comparisons
x! = 5!: y! = 3.333
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF x! < y! THEN x = 1
  NEXT j
```

```
NEXT i
t!(24) = TIMER - t! - t!(0)

'time for 1,000,000 double-precision
     comparisons
x# = 5#: y# = 3.333#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF x# < y# THEN x = 1
  NEXT j
NEXT i
t!(25) = TIMER - t! - t!(0)

'is there short-circuit expression
     evaluation?
'integer loop, 1,000,000 times
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF i < 0 AND j < 10 THEN x = 1
  NEXT j
NEXT i
t!(26) = TIMER - t! - t!(0)
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF j < 10 AND i < 0 THEN x = 1
  NEXT j
NEXT i
t!(27) = TIMER - t! - t!(0)

'screen output: print 1,000 70-byte strings
x$ = STRING$(70, 66)
t! = TIMER
FOR i = 1 TO 1000
  PRINT x$
NEXT i
t!(28) = TIMER - t! - t!(0) / 1000

'print results of benchmark
PRINT #1, "Empty integer loop, 1,000,000
     iterations:"; TAB(46); t!(0)
PRINT #1, "1,000,000 integer
     assignments:"; TAB(46); t!(1)
PRINT #1, "1,000,000 integer
     additions:"; TAB(46); t!(2)
PRINT #1, "1,000,000 integer
     subtractions:"; TAB(46); t!(3)
PRINT #1, "1,000,000 integer
     multiplications:"; TAB(46); t!(4)
PRINT #1, "1,000,000 integer
     divisions:"; TAB(46); t!(5)
PRINT #1, "100,000 string
     assignments:"; TAB(46); t!(6)
PRINT #1, "100,000 string MID$
     operations:"; TAB(46); t!(7)
PRINT #1, "10,000 string
     concatenations:"; TAB(46); t!(8)
PRINT #1, "1,000,000 single-precision
     assignments:"; TAB(46); t!(9)
PRINT #1, "1,000,000 single-precision
     additions:"; TAB(46); t!(10)
PRINT #1, "1,000,000 single-precision
     subtractions:"; TAB(46); t!(11)
PRINT #1, "100,000 single-precision
     multiplications:"; TAB(46); t!(12)
PRINT #1, "100,000 single-precision
     divisions:"; TAB(46); t!(13)
PRINT #1, "Error in 100,000 single-
   precision mult/div:"; TAB(46); t!(14)
PRINT #1, "10,000 single-precision
     exponentiations:"; TAB(46); t!(15)
PRINT #1, "1,000,000 double-precision
     assignments:"; TAB(46); t!(16)
PRINT #1, "1,000,000 double-precision
     additions:"; TAB(46); t!(17)
PRINT #1, "1,000,000 double-precision
     subtractions:"; TAB(46); t!(18)
PRINT #1, "100,000 double-precision
     multiplications:"; TAB(46); t!(19)
PRINT #1, "100,000 double-precision
     divisions:"; TAB(46); t!(20)
PRINT #1, "Error in 100,000 double-
   precision mult/div:"; TAB(46); t!(21)
PRINT #1, "10,000 double-precision
     exponentiations:"; TAB(46); t!(22)
PRINT #1, "1,000,000 integer
     comparisons:"; TAB(46); t!(23)
PRINT #1, "1,000,000 single-precision
     comparisons:"; TAB(46); t!(24)
PRINT #1, "1,000,000 double-precision
     comparisons:"; TAB(46); t!(25)
PRINT #1, "1,000,000 conditional
  integer assignments:"; TAB(46); t!(26)
PRINT #1, "1,000,000 conditional
 assignments (reversed):"; TAB(46); t!(27)
PRINT #1, "Print 1,000 70-byte
  strings to the screen:"; TAB(46); t!(28)
END
```

End Listing Three

Listing Four

```
DEFINT A-Z
DIM t!(28)
OPEN "bas6.tim" FOR OUTPUT AS #1
'time for a raw integer loop,
     executed 1,000,000 times.
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for a integer assignment loop,
     executed 1,000,000 times.
y = 5: z = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 integer adds
y = 5: z = -5
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x + y
  x = x + z
  NEXT j
NEXT i
t!(2) = (TIMER - t! - t!(0)) / 2 - t!(1)

'time for 1,000,000 integer subtracts
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x - y
  x = x - z
  NEXT j
NEXT i
t!(3) = (TIMER - t! - t!(0)) / 2 - t!(1)

'time for 1,000,000 integer multiplies
k = 7
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = k * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000 integer divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'time for 100,000 string assignments
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x$ = "abcdefghijklmnopqrstuvwxyz"
  x$ = "zyxwvutsrqponmlkjihgfedcba"
  NEXT j
NEXT i
t!(6) = (TIMER - t! - t!(0) / 10) / 2

'time for 100,000 string MID$ operations
x$ = "abcdefghijklmnopqrstuvwxyz"
k = 17
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  MID$(x$, k, 1) = "d"
  NEXT j
  NEXT i
  t!(7) = TIMER - t! - t!(0) / 10

  'time for 10,000 string concatenations
  x$ = ""
  t! = TIMER
  FOR i = 1 TO 10000
     x$ = x$ + "a"
  NEXT i
  t!(8) = TIMER - t! - t!(6) - t!(0) / 100

  'time for a single-precision assignment
       loop, executed 1,000,000 times.
  y! = 5!: z! = -5!
  t! = TIMER
  FOR i = 1 TO 1000
    FOR j = 1 TO 100
    x! = y!
    x! = z!
    NEXT j
  NEXT i
  t!(9) = (TIMER - t! - t!(0)/10) / 2

  'time for 1,000,000 single-precision adds
  t! = TIMER
  FOR i = 1 TO 1000
    FOR j = 1 TO 100
      x! = x! + y!
      x! = x! + z!
    NEXT j
  NEXT i
  t!(10) = (TIMER - t! - t!(0)/10) / 2 - t!(9)

  'time for 1,000,000 single-precision
       subtracts
  x! = 0!
  t! = TIMER
  FOR i = 1 TO 1000
    FOR j = 1 TO 100
      x! = x! - y!
      x! = x! - z!
    NEXT j
  NEXT i
  t!(11) = (TIMER - t! - t!(0)/10) / 2 - t!(9)

  'time for 100,000 single-precision
       multiplies
  x! = 1!
  t! = TIMER
  FOR i = 1 TO 1000
    FOR j = 1 TO 100
      x! = x! * 1.00001
    NEXT j
  NEXT i
  t!(12) = (TIMER - t! - t!(0) / 10) / 2 - t!(9)

  'time for 100,000 single-precision divides
  t! = TIMER
  FOR i = 1 TO 1000
    FOR j = 1 TO 100
      x! = x! / 1.00001
    NEXT j
  NEXT i
  t!(13) = (TIMER - t! - t!(0) / 10) / 2 - t!(9)

  'error in single-precision multiply/divide
  t!(14) = x! - 1!
```

```
'time for 10,000 single-precision
      exponentiations
x! = 100!
t! = TIMER
FOR i = 1 TO 1000
  x! = x! ^ .999999
NEXT i
t!(15) = (TIMER - t! - t!(0) / 1000) / 2 - t!(9) / 100

'time for a double-precision assignment
      loop, executed 1,000,000 times.
y# = 5.5#: z# = -5.5#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x# = y#
  x# = z#
  NEXT j
NEXT i
t!(16) = (TIMER - t! - t!(0)/10) / 2

'time for 1,000,000 double-precision adds
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# + y#
  NEXT j
NEXT i
t!(17) = TIMER - t! - t!(16) - t!(0)/10

'time for 1,000,000 double-precision subtracts
x# = 0#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# - y#
  NEXT j
NEXT i
t!(18) = TIMER - t! - t!(16) - t!(0)/10

'time for 100,000 double-precision multiplies
x# = 1#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
    x# = x# * 1.00001#
  NEXT j
NEXT i
t!(19) = (TIMER - t! - t!(0) / 100) - t!(16) / 10

'time for 100,000 double-precision divides
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 10
    x# = x# / 1.00001#
  NEXT j
NEXT i
t!(20) = (TIMER - t! - t!(0) / 100) - t!(16) / 10

'error in double-precision multiply/divide
t!(21) = x# - 1#

'time for 10,000 double-precision
      exponentiations
x# = 100#
t! = TIMER
FOR i = 1 TO 1000
  x# = x# ^ .999999#
NEXT i
t!(22) = (TIMER - t! - t!(0) / 1000) - t!(16) / 100

'following are logical comparisons
      and operators

'time for 1,000,000 integer comparisons
x = 0
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    IF i < x THEN x = 1
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)/10

'time for 1,000,000 single-precision comparisons
x! = 5!: y! = 3.333
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    IF x! < y! THEN x = 1
  NEXT j
NEXT i
t!(24) = TIMER - t! - t!(0)/10

'time for 1,000,000 double-precision comparisons
x# = 5#: y# = 3.333#
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    IF x# < y# THEN x = 1
  NEXT j
NEXT i
t!(25) = TIMER - t! - t!(0)/10

'is there short-circuit expression evaluation?
'integer loop, 1,000,000 times
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    IF i < 0 AND j < 10 THEN x = 1
  NEXT j
NEXT i
t!(26) = TIMER - t! - t!(0)/10
t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    IF j < 10 AND i < 0 THEN x = 1
  NEXT j
NEXT i
t!(27) = TIMER - t! - t!(0)/10

'screen output: print 1,000 70-byte strings
x$ = STRING$(70, 66)
t! = TIMER
FOR i = 1 TO 1000
  PRINT x$
NEXT i
t!(28) = TIMER - t! - t!(0) / 1000

'print results of benchmark
PRINT #1, "Empty integer loop, 1,000,000
      iterations:"; TAB(46); t!(0)
PRINT #1, "1,000,000 integer
      assignments:"; TAB(46); t!(1)
PRINT #1, "1,000,000 integer
      additions:"; TAB(46); t!(2)
PRINT #1, "1,000,000 integer
      subtractions:"; TAB(46); t!(3)
PRINT #1, "1,000,000 integer
      multiplications:"; TAB(46); t!(4)
PRINT #1, "1,000,000 integer
      divisions:"; TAB(46); t!(5)
PRINT #1, "100,000 string
      assignments:"; TAB(46); t!(6)
PRINT #1, "100,000 string MID$
      operations:"; TAB(46); t!(7)
PRINT #1, "10,000 string
      concatenations:"; TAB(46); t!(8)
PRINT #1, "1,000,000 single-precision
      assignments:"; TAB(46); t!(9)*10
PRINT #1, "1,000,000 single-precision
      additions:"; TAB(46); t!(10)*10
PRINT #1, "1,000,000 single-precision
      subtractions:"; TAB(46); t!(11)*10
PRINT #1, "100,000 single-precision
      multiplications:"; TAB(46); t!(12)
PRINT #1, "100,000 single-precision
      divisions:"; TAB(46); t!(13)
PRINT #1, "Error in 100,000 single-precision
      mult/div:"; TAB(46); t!(14)
PRINT #1, "10,000 single-precision
      exponentiations:"; TAB(46); t!(15)*10
PRINT #1, "1,000,000 double-precision
      assignments:"; TAB(46); t!(16)*10
PRINT #1, "1,000,000 double-precision
      additions:"; TAB(46); t!(17)*10
PRINT #1, "1,000,000 double-precision
      subtractions:"; TAB(46); t!(18)*10
PRINT #1, "100,000 double-precision
      multiplications:"; TAB(46); t!(19)*10
PRINT #1, "100,000 double-precision
      divisions:"; TAB(46); t!(20)*10
PRINT #1, "Error in 100,000 double-precision
      mult/div:"; TAB(46); t!(21)*10
PRINT #1, "10,000 double-precision
      exponentiations:"; TAB(46); t!(22)*10
PRINT #1, "1,000,000 integer
      comparisons:"; TAB(46); t!(23)*10
PRINT #1, "1,000,000 single-precision
      comparisons:"; TAB(46); t!(24)*10
PRINT #1, "1,000,000 double-precision
      comparisons:"; TAB(46); t!(25)*10
PRINT #1, "1,000,000 conditional
      integer assignments:"; TAB(46); t!(26)*10
PRINT #1, "1,000,000 conditional
      assignments (reversed):"; TAB(46); t!(27)*10
PRINT #1, "Print 1,000 70-byte strings
     to the screen:"; TAB(46); t!(28)
END
```

End Listings

Listing One *(Text begins on page 116.)*

```
#include <stdio.h>
#include <ctype.h>
#include <tools/getargs.h>

/*------------------------------------------------------------------
 * SHELLARC.C   This program manufactures a shell archive.
 *              Usage is:
 *                      shellarc [-ffile] [-q] names...
 *
 * if -f is specified, the rest of the command line is ignored
 * and file names are taken from the indicated file, otherwise
 * filenames are taken from the command line.
 * The output archive looks like this:
 *
 *      cat >file <<"END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *      <contents of file>
 *      "END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *
 * Lines (or command-line arguments) that start with a = are
 * written to the output file in the corresponding place.
 * For example, the list:
 *
 *              file1
 *              = mkdir foo
 *              foo/file2
 *
 * comes out as follows:
 *
 *   cat >file1 <<"END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *   <contents of file1>
 *   "END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *   mkdir foo
 *   cat >foo/file2 <<"END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *   <contents of foo/file2>
 *   "END-OF-FILE--END-OF-FILE--END-OF-FILE"
 *
 * The = sign and any trailing whitespaces is removed.
 *
 * If -q is specified, the quotes are omitted from the
 * END-OF-FILE... In this case the shell will expand shell
 * varaibles, etc, that are present in the file.
 *------------------------------------------------------------------
 */

char    **Argv  = NULL ;
int     Argc    = 0 ;
FILE    *Fd     = NULL ;
int     Noquote = 0 ;
void    get_file();
int     Dodot = 0;

#define NOQUOTE_MARK "END-OF-FILE--END-OF-FILE--END-OF-FILE"
#define NORM_MARK    "\"END-OF-FILE--END-OF-FILE--END-OF-FILE\""
#define BUFSIZE      1024

ARG     Argtab[] =
{
{'f',PROC,    (int *)get_file,"Use <str> as command file"        },
{'q',BOOLEAN,&Noquote,        "Expand shell vars on dearchiving" },
{'.',BOOLEAN,&Dodot,          "precede all file names with a dot"}
};

static char *Usage_msg[] =
{
"",
"Shellarc makes a shell archive by concatenating together the",
"files listed on the command line (or in the command file) with",
"interspersed shell directives so that, when the output file is",
"run, it will unpack itself. For example, shellarc \"=mkdir test\"",
"test/file1 test/file2 sends the following to standard output:",
"",
"\tmkdir test",
"\tcat >test/file1 <<\"END-OF-FILE--END-OF-FILE--END-OF-FILE\"",
"\t<contents of test/file1>",
"\t\"END-OF-FILE--END-OF-FILE--END-OF-FILE\"",
"\tcat >test/file2 <<\"END-OF-FILE--END-OF-FILE--END-OF-FILE\"",
"\t<contents of test/file2>",
"\t\"END-OF-FILE--END-OF-FILE--END-OF-FILE\"",
"",
"Arguments that begin with = are just written to the output",
"file in the place coresponding to the position on the command",
"line (with the = stripped. -q removes the quotes from the",
"END-OF-FILE... string. -f causes commands to be taken from the",
"file instead of the command line, one file name or = command",
"per line",
NULL
};

/*------------------------------------------------------------------*/

usage()
{
    char **p;
    for( p = Usage_msg; *p; puts(*p++) )
        ;

    exit( 1 );
}

/*------------------------------------------------------------------*/

void    get_file( name )
char    *name;
{
    /* Open a file and exit if you can't */

    if( !(Fd = fopen(name,"r") ))
    {
        perror( name );
        exit( 1 );
    }
}

/*------------------------------------------------------------------*/

char    *nextline( buf, n )
char    *buf;
{
    /* Get the next input line of the description file. Get the
     * line either from a file specified with -f on the command
     * line (if Fd is NULL), or from the command line itself (if
     * it's not). Return NULL at ten of file, the buf argument
     * otherwise.
     */

    char *rval;

    if( Fd )
    {
        if( rval = fgets( buf, n, Fd ) )
            buf[ strlen(buf) - 1 ] = '\0' ; /* Overwrite \n */

        return rval;
    }
    else if( --Argc )
    {
        strncpy( buf, *++Argv, n );
        return buf;
    }
    else
        return NULL;
}

/*------------------------------------------------------------------*/

main( argc, argv )
char    **argv;
{
    char        *mark = NORM_MARK ;
    FILE        *ifile;
    static char buf[ BUFSIZE ];
    char        *p;
    char        *dot = "";

    reargv( &argc, &argv );

    Argc = getargs( argc, argv, Argtab,
                        sizeof(Argtab)/sizeof(ARG), usage );
    Argv = argv;

    if( Noquote )
        mark = NOQUOTE_MARK ;

    if( Dodot )
        dot = "." ;

    printf("#\n");              /* Make this a C-shell script */

    while( nextline( buf, BUFSIZE ) )
    {
        if( *buf == '=' )
        {
            for(p = buf + 1; isspace( *p ) ; ++p )
                ;
            puts( p );
        }
        else
        {
            if( !(ifile = fopen(buf, "r")) )
                perror( buf );
            else
            {
                fprintf( stderr, "%s:\n", buf );
                printf ("echo Unarchiving %s%s\n", dot, buf );
                printf ("cat >%s%s <<%s\n",   dot, buf, mark );

                while( fgets(buf, BUFSIZE, ifile) )
                    fputs( buf, stdout );

                fclose( ifile );
                puts( mark );
            }
        }
    }
}
```

End Listing One

Listing Two

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <io.h>             /* prototype for mktemp()           */

#define MAXARRAY 256        /* Maximum input line size          */
#define REDIR   0           /* return value from extract_stuff() */
#define NOREDIR 1           /* "                                */

/*------------------------------------------------------------------*/

char    *do_input( word, echo, tname, inp )
char    *word;              /* Terminal string              */
int     echo;               /* echo file to stdout if true  */
FILE    *inp;               /* input stream                 */
{
    /* Read lines from "inp" up to "word" into a temporary
     * file, then close the temporary file, and return its name.
     */

    char *temp_name;
    static char template[ 16 ];

    FILE *tfile;            /* FILE pointer to temporary file */
    char buf[256];          /* Input line buffer.             */
```

```
    strcpy(template,"eaXXXXXX");
    if( (temp_name = mktemp(template)) == NULL )
    {
        fprintf(stderr,"Can't make temporary file name\n");
        exit(1);
    }

    if( !(tfile = fopen( temp_name, "w" )) )
    {
        perror( temp_name );
        exit(1);
    }

    while( fgets(buf, sizeof(buf), inp) )
    {
        if( strcmp(buf, word) == 0 )
            break;

        if( echo )
            fprintf( stderr, "<< %s", buf );

        fputs( buf,  tfile );
    }

    fclose( tfile );
    return temp_name ;
}

/*------------------------------------------------------------*/

usage()
{
    fprintf(stderr, "Usage:     exp-arc [-e] archive...\n\n");
    fprintf(stderr, "Expand a UNIX shell archive.\n");
    fprintf(stderr, "-e to print expanded files to stderr\n");
    exit( 1 );
}

/*------------------------------------------------------------*/

main( argc, argv )
char    **argv;
{
    /* Usage:   exp-arc [-e] archive_file...
     *
     *  Expand a UNIX shell archive.
     *  -e to print files to standard error as they're expanded
     */

    int         echo = 0;
    int         i;
    char        *temp_name;
    FILE        *archive;
    char        *word;
    static char command[ MAXARRAY ];

    ++argv;
    --argc;

    if( argc && argv[0][0] == '-' )
    {
        if( argv[0][1] == 'e' )
            echo = 1;
        else
            usage();

        --argc ;
        ++argv ;
    }

    if( argc == 0 )
        usage();

    for( ; --argc >= 0; ++argv )
    {
        if( !(archive = fopen(*argv, "r")) )
        {
            perror( *argv );
            exit(1);
        }

        while((i = extract_stuff(command,&word,archive)) != EOF)
        {
            if( i == NOREDIR )
                system( command );
            else
            {
                temp_name=do_input(word,echo,temp_name,archive);

                if( !freopen( temp_name, "r", stdin )  )
                {
                    perror( temp_name );
                    exit(1);
                }

                system( command );

                freopen( "con:", "r", stdin );
                unlink ( temp_name );
            }
        }

        fclose( archive );
    }
}

/*------------------------------------------------------------*/

int     extract_stuff( command, word, inp )
char    *command;
char    **word;
FILE    *inp;
{
    /* Read a line of input. If it takes the form:
     *
     *          cat >file <<pattern
     *
     * put everything up to the << into "command."
     * Modify *word to point at word. That is,
     * given the previous line, command will
     * hold "cat >file" and word will hold
     * "pattern". The space to the right of
     * the 'e' in file will be replaced with a
     * '\0'. Return REDIR in this case.
     * If the "cat," the >, or the << is
     * missing, put the entire line into command
     * and return NOREDIR. Return EOF at end of
     * file.
     */

    char        *p, *endp;

    if( ! fgets(command,MAXARRAY,inp) )
        return EOF;

    for( p = command; isspace(*p) ; ++p )
        ;

    if( !(p[0]=='c' && p[1]=='a' && p[2]=='t' ) )
        return NOREDIR;

    for( p += 3; isspace(*p) ; ++p )
        ;

    if( p[0] != '>' )
        return NOREDIR;

    for( ++p; isspace(*p); ++p )
        ;

    while( *p && !isspace(*p) )
        ++p;
    endp = p;                   /* Just remember it for now */

    for(; isspace(*p); ++p )
        ;

    if( !(p[0]=='<' && p[1]=='<') )
        return NOREDIR;

    for( p += 2 ; isspace(*p); ++p )
        ;

    *endp = '\0';               /* Now terminate it */
    *word = p ;
    return REDIR;
}
```

End Listings

STRUCTURED PROGRAMMING

Listing One *(Text begins on page 122)*

```
UNIT popups;

(* Kent Porter, DDJ, July '88 issue          *)
(* Support for pop-up windows and menu bars *)
(* Works with MDA, Compaq, CGA, EGA, VGA     *)
(* Turbo Pascal 4.0                          *)

INTERFACE

USES dos, crt;

(* These are names for common keystrokes *)

CONST  F1          = #187;                    { Second byte plus 128 }
       HomeKey     = #199;
       EndKey      = #207;
       PgUp        = #201;
       PgDn        = #209;
       UpCursor    = #200;
       DownCursor  = #208;
       LeftCursor  = #203;
       RiteCursor  = #205;
       Enter       =  #13;

(* These are structures used by the routines *)

CONST  SEP = '';               { Element separator in menu contents }

TYPE
  strPtr = ^STRING;
  popRec = RECORD
    left, top, right, bottom,                    { Border locations }
      style,                                       { Border style }
      normal, hilite,                            { Text attributes }
      normback, hiback, border : INTEGER;
    contents : strPtr;                         { Fixed text contents }
    save : POINTER;                  { pointer to display save buffer }
    oldMin, oldMax : WORD;             { Previous window dimensions }
    oldX, oldY : INTEGER;                { previous cursor location }
    oldColor : WORD;                { previous fore/background colors }
  END;

  menuRec = RECORD
    row,                                       { row where bar appears }
      interval,                              { cols between first chars }
      fore, back : INTEGER;                  { fore/background colors }
    choice : strPtr;                         { pointer to text contents }
  END;
```

```
VAR VideoBuffer : POINTER;      { Global pointer to text video buffer }

(* List of exported routines in this module *)
(* ---------------------------------------- *)

PROCEDURE textbox (left, top, right, bottom, style : INTEGER);
PROCEDURE popShow (VAR pop : popRec);
PROCEDURE popErase (VAR pop : popRec);
PROCEDURE popCenter (VAR pop : popRec; row : INTEGER; info : STRING);
PROCEDURE popHilite (VAR pop : popRec; row : INTEGER);
PROCEDURE popNormal (VAR pop : popRec; row : INTEGER);
PROCEDURE showMenubar (VAR spec : menuRec);
PROCEDURE cursOff;
PROCEDURE cursOn;
FUNCTION  Keystroke : CHAR;

(* ------------------------------------------------------------------ *)

IMPLEMENTATION

{ Private identifiers }

CONST bufSize = 4096;                          { size of video buffer }
      border : ARRAY [1..2, 0..5] OF CHAR =       { box border chars }
          (( #196, #179, #218, #191, #217, #192),
           ( #205, #186, #201, #187, #188, #200));

VAR   egaByte : WORD ABSOLUTE $0040:$0087;          { EGA eqpt byte }
      reg     : REGISTERS;                  { regs for low-level calls }
      mode    : WORD;                          { current video mode }

{ Routine bodies follow }

PROCEDURE textbox;

     { Draw textbox in indicated style, where:
          0 = no border
          1 = single score
          2 = double score }

VAR  r, c : INTEGER;

BEGIN
  IF style IN [1..2] THEN BEGIN

    { Draw horizontals }
    FOR c := (left+1) TO right DO BEGIN
      Gotoxy (c, top);       WRITE (border [style, 0]);
      Gotoxy (c, bottom);    WRITE (border [style, 0]);
    END;


{ Draw verticals }
FOR r := (top+1) TO bottom DO BEGIN
      Gotoxy (left, r);      WRITE (border [style, 1]);
      Gotoxy (right, r);     WRITE (border [style, 1]);
    END;

    { Draw corners }
    Gotoxy (left, top);      WRITE (border [style, 2]);
    Gotoxy (right, top);     WRITE (border [style, 3]);
    Gotoxy (right, bottom); WRITE (border [style, 4]);
    Gotoxy (left, bottom);  WRITE (border [style, 5]);
  END;
END;       { of textbox }

(* ---------------------------- *)

PROCEDURE popShow;

     { display popup described by passed structure }

  PROCEDURE popWrite (VAR winText : STRING);

      { Local proc to write fixed popup contents, if any }

  VAR  p : INTEGER;

  BEGIN
    IF pop.contents <> NIL THEN BEGIN
      GOTOXY (2, 1);                  { Always leave 1 leading space }
      FOR p := 1 TO length (winText) DO
        IF winText [p] <> SEP THEN
          WRITE (winText [p])
        ELSE
          GOTOXY (2, whereY + 1);       { Go to next row on separator }
    END;
  END;   { of popWrite }

BEGIN { Body of popShow }

  { Get the current video state }
  pop.oldMin := windMin + $0101;
  pop.oldMax := windMax + $0101;                  { window dimensions }
  pop.oldColor := textAttr;                         { current colors }
  pop.oldX := whereX; pop.oldY := whereY;          { cursor position }
  Window (1, 1, 80, 25);                { reset window to full screen }

  { Save the current screen }
  GetMem (pop.save, bufSize);                { allocate space for it }
  Move (videoBuffer^, pop.save^, bufSize);              { save screen }

  { Draw the border for the popup }
  WITH pop DO BEGIN
    Textcolor (border);
    Textbackground (normback);
    Textbox (left, top, right, bottom, style);

  { Open the window }
    Textcolor (normal);
    Window (left+1, top+1, right-1, bottom-1);
  END;   { of WITH }

  { Write fixed text }
  ClrScr;
  popWrite (pop.contents^);
END;

(* ---------------------------- *)

PROCEDURE popErase;

      { Erase pop-up window, restoring overlaid image }

BEGIN

  { Make sure there's a saved image to restore }
  IF pop.save <> NIL THEN BEGIN
    window (1, 1, 80, 25);

  { Restore previous video state }
    WITH pop DO BEGIN
      Window (LO (oldMin), HI (oldMin),
              LO (oldMax), HI (oldMax));
      Textcolor (oldColor AND $0F);
      TextBackground (oldColor SHR 4);
      Gotoxy (pop.oldX, pop.oldY);
    END;

  { Restore overlaid screen image }
    Move (pop.save^, videoBuffer^, bufSize);
    FreeMem (pop.save, bufSize);
    pop.save := NIL;
  END;
END;

(* ---------------------------- *)

PROCEDURE popCenter;

      { Center string in window at specified row }

VAR   col : INTEGER;

BEGIN

IF pop.save <> NIL THEN                          { pop-up is visible }
  IF row < pop.bottom - pop.top THEN BEGIN          { row is legal }

      col := (pop.right - pop.left - Length (info)) DIV 2;
      Gotoxy (col, row);
      WRITE (info);
    END;
END;

(* ---------------------------- *)

PROCEDURE popRewrite (VAR pop : popRec; row : INTEGER; attrib : BYTE);

      { Local proc called by popHilite and popNormal     }
      { Rewrites pop-up row with new character attribute }

VAR  p, nchars : INTEGER;

BEGIN

  IF pop.save <> NIL THEN                          { pop-up is visible }
    IF row < pop.bottom - pop.top THEN BEGIN
      nchars := pop.right - pop.left - 1;          { Get width of row }
      FOR p := 1 TO nchars DO BEGIN       { For each char in row do.. }
        Gotoxy (p, row);                                 { goto char }
        reg.ah := 8;                                      { Get char }
        reg.bh := 0;
        intr (16, reg);                                { via ROM BIOS }
        reg.ah := 9;                                { write back out with }
        reg.bl := attrib;                              { hilite attribs }
        reg.bh := 0;
        reg.cx := 1;
        intr (16, reg);
      END;
    END;
END;

(* ---------------------------- *)

PROCEDURE popHilite;

      { Highlight text in specified pop-up row }

VAR   attrib : BYTE;
      x, y   : INTEGER;

BEGIN
  x := whereX; y := whereY;                     { Save cursor position }
  Attrib := pop.hilite + (pop.hiback SHL 4);    { Set text attributes }
  popRewrite (pop, row, attrib);                        { Rewrite row }
  gotoxy (x, y);                                { Restore cursor }
END;

(* ---------------------------- *)

PROCEDURE popNormal;

      { Set text in pop-up row to normal attributes }

VAR   attrib : BYTE;
      x, y   : INTEGER;

BEGIN
  x := whereX; y := whereY;
  Attrib := pop.normal + (pop.normback SHL 4);
  popRewrite (pop, row, attrib);
  gotoxy (x, y);
END;

PROCEDURE menuBar;
BEGIN
END;

(* ---------------------------- *)

PROCEDURE showMenubar;

      { Place menu bar in current window }
```

```
VAR    p, c, color, curX, curY : INTEGER;
       x1, x2                  : INTEGER;

BEGIN

  { Save video state information }
  curX := whereX; curY := whereY;
  color := TextAttr;
  x1 := Lo (WindMin);
  x2 := Lo (WindMax);

  { Set colors for menu }
  TextColor (spec.fore);
  TextBackground (spec.back);
  gotoxy (1, spec.row);
  WRITELN (' ');

  { Write out the bar background first }
  Gotoxy (1, spec.row);
  FOR p := x1 TO x2 DO
    WRITE (' ');

  { Write the menubar text }
  Gotoxy (1, spec.row);                          { First item location }
  c := 1;                                          { Item counter }
  FOR p := 1 TO Length (spec.choice^) DO BEGIN     { Char by char }
    IF spec.choice^[p] <> SEP THEN                 { If not delim, }
      WRITE (spec.choice^[p])                        { write char }
    ELSE BEGIN                                       { else }
      Gotoxy ((spec.interval * c) + 1, spec.row);  { Go to next item }
      INC (c);                                       { Count items }
    END
  END;

  { Restore video state }
  TextColor (color AND $0F);
  TextBackground (color SHR 4);
  Gotoxy (curX, curY);
END;

(* -------------------------- *)

PROCEDURE cursOff;

      { Turn off hardware cursor }

BEGIN
  reg.ah := 3;                              { get current cursor shape }
  reg.bh := 0;                            { NOTE: works in page 0 only }
  Intr (16, reg);
  reg.ch := reg.ch OR $20;                              { turn on bit 5 }
  reg.ah := 1;
  Intr (16, reg);                                         { tell BIOS }
END;

(* -------------------------- *)

PROCEDURE cursOn;

      { Turn hardware cursor back on }

BEGIN
  reg.ah := 3;                                       { As above, except }
  reg.bh := 0;
  Intr (16, reg);
  reg.ch := reg.ch AND $DF;                           { turn off bit 5 }
  reg.ah := 1;
  Intr (16, reg);
END;

(* -------------------------- *)

FUNCTION Keystroke;

      { Wait for a keystroke. If it's a special key (0+code), }
      { return the second byte + 128, else return upper case  }

VAR   ch : CHAR;

BEGIN
  ch := UpCase (ReadKey);                               { Get keystroke }
  IF ch = chr (0) THEN BEGIN                      { if a lead-in, then... }
    ch := ReadKey;                              { get the second byte and }
    ch := chr (ord (ch) + 128);                         { shift up by 128 }
  END;
  Keystroke := ch;
END;

(* ------------------------------------------------------------------ *)

{ INITIALIZATION CODE SETS ADDRESS OF VIDEO BUFFER }

Begin
    Reg.ah := 15;                               { Get current video mode }
    Intr (16, reg);
    mode := reg.al;

    IF (mode = 7) OR (mode = 2) THEN       { Either MDA or Compaq MDA }
      videoBuffer := ptr ($B000, $0000)
    ELSE
      videoBuffer := ptr ($B800, $0000);          { else color buffer }
END.     { of unit POPUPS.PAS }
```

End Listing One

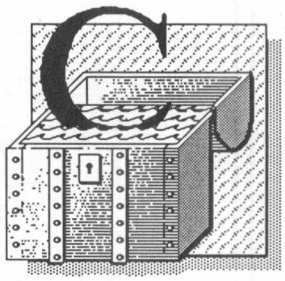
Listing Two

```
Program popdemo;

  (* Demonstrates use of POPUPS unit *)

USES crt, popups;

(* Define menubar *)

CONST MainMenu : menuRec = (
            row : 1; interval : 26; fore : White; back : Green);
      MainMenuText : string [14] = 'DemosRunThru';

(* Define pull-down menus *)

      DemoMenu : popRec = (
            left : 1; top : 2; right : 14; bottom : 6; style : 1;
            normal : LightGray; hilite : LightGray;
            normback : Blue; hiback : Magenta; border : Cyan);
      DemoMenuText : string [39] =
            'Show fileWalk popupGraphics';
      RunMenu : popRec = (
            left : 27; top : 2; right : 39; bottom : 5; style : 1;
            normal : LightGray; hilite : LightGray;
            normback : Blue; hiback : Magenta; border : Cyan);
      RunMenuText : string [19] = 'All demosExit menu';

      QuitMenu : popRec = (
            left : 53; top : 2; right : 68; bottom : 5; style : 1;
            normal : LightGray; hilite : LightGray;
            normback : Blue; hiback : Magenta; border : Cyan);
      QuitMenuText : string [22] = 'Quit programExit menu';

(* Define windows for demos *)
      FileWindow : popRec = (
            left : 2; top : 4; right : 75; bottom : 25; style : 2;
            normal : Brown; hilite : 0;
            normback : Black; hiback : 0; border : Yellow);

      Walker : popRec = (
            left : 66; top : 1; right : 80; bottom : 6; style : 2;
            normal : Red; hilite : 0;
            normback : LightGray; hiback : 0; border : Red);
      WalkerText : string [46] =
            'This pop-upmoves downby changingtop, bottom';

      SalesChart : popRec = (
            left : 51; top : 9; right : 72; bottom : 22; style : 2;
            normal : Cyan; hilite : 0;
            normback : LightGray; hiback : 0; border : Red);

(* ------------------------ DEMO ROUTINES -------------------------- *)

PROCEDURE ShowFile;

      { List the source for this program in FileWindow }
      { Leave window open afterwards }

VAR   ThisFile : TEXT;
      Line     : string [80];

BEGIN
  popShow (FileWindow);
  Assign (ThisFile, 'POPDEMO.PAS');
  Reset (ThisFile);
  WHILE NOT eof (ThisFile) DO BEGIN
    Readln (ThisFile, line);
    Writeln (Line);
  END;
  Close (ThisFile);
END;

(* -------------------------- *)

PROCEDURE WalkPopup;

      { Walk a pop-up down the right side of the display }
      { Moves by successively incrementing top and bottom }

VAR   TopRow : INTEGER;

BEGIN
  popShow (Walker);
  FOR TopRow := 2 to 19 DO BEGIN
    popErase (Walker);
    Walker.top    := TopRow;
    Walker.bottom := TopRow + 5;
    popShow (Walker);
  END;
END;

(* -------------------------- *)

PROCEDURE TextChart;

      { Simulate a sales results chart using simple text graphics }

CONST block = #219;

  PROCEDURE DrawBar (column, height : INTEGER);
  VAR   y : INTEGER;
  BEGIN
    FOR y := 10 DOWNTO (10 - height) DO BEGIN
      Gotoxy (column, y);
      Write (block);
    END;
  END;

BEGIN
  popShow (SalesChart);
  popCenter (SalesChart, 1, 'Sales Results');
  TextColor (Green);
  Gotoxy (2, 12); Write ('Projected');
  DrawBar ( 2, 6);
  DrawBar ( 7, 7);
  DrawBar (12, 5);
  DrawBar (17, 6);
  TextColor (Red);
  Gotoxy (14, 12); Write ('Actual');
  DrawBar ( 4, 5);
  DrawBar ( 9, 6);
  DrawBar (14, 7);
  DrawBar (19, 8);
END;
```

```
(* ---------------------- CONTROL ROUTINES ---------------------- *)

FUNCTION DemoResult : CHAR;

    { Pull down and act on Demos Menu }

VAR   key, wait : CHAR;
      pick      : INTEGER;
      exiting   : BOOLEAN;

BEGIN
  pick := 1;
  exiting := FALSE;
  popShow (DemoMenu);
  popHilite (DemoMenu, 1);
  REPEAT

    { Get menu selection }
    key := Keystroke;
    popNormal (DemoMenu, pick);                    { remove hilite bar }
    CASE key OF
      'S'         : pick := 1;
      'W'         : pick := 2;
      'G'         : pick := 3;
      DownCursor : BEGIN
                     Inc (pick);
                     IF pick > 3 THEN pick := 1;   { Wrap to top row }
                   END;
      UpCursor   : BEGIN
                     Dec (pick);
                     IF pick = 0 THEN pick := 3;    { Wrap to bottom }
                   END;
      LeftCursor : exiting := TRUE;
      RiteCursor : exiting := TRUE;
      Enter      : CASE pick OF        { Selection by cursor + enter }
                     1: key := 'S';
                     2: key := 'W';
                     3: key := 'G';
                   END;
      ELSE exiting := TRUE;                      { Pass back keystroke }
    END;
    popHilite (DemoMenu, pick);                    { hilite new pick }

    { Do demo if selected }
    IF key IN ['S', 'W', 'G'] THEN BEGIN
      CASE key OF
        'S': BEGIN
               ShowFile;
               wait := ReadKey;
               popErase (FileWindow);
             END;
        'W': BEGIN
               WalkPopup;
               wait := ReadKey;
               popErase (Walker);
             END;
        'G': BEGIN
               TextChart;
               wait := ReadKey;
               popErase (SalesChart);
             END;
      END;
    END;
  UNTIL exiting;
  popErase (DemoMenu);
  DemoResult := key;
END;

(* ---------------------------- *)

FUNCTION RunResult : CHAR;

      { Pull down and act on Run Menu }

VAR   key, wait : CHAR;
      pick      : INTEGER;
      exiting   : BOOLEAN;

BEGIN
  pick := 1;
  exiting := FALSE;
  popShow (RunMenu);
  popHilite (RunMenu, 1);
  REPEAT
    key := Keystroke;                              { remove hilite }
    popNormal (RunMenu, pick);
    CASE key OF
      DownCursor : IF pick = 1 THEN pick := 2 ELSE pick := 1;
      UpCursor   : IF pick = 1 THEN pick := 2 ELSE pick := 1;
      'E'        : exiting := TRUE;
      LeftCursor : exiting := TRUE;
      RiteCursor : exiting := TRUE;
      Enter      : IF pick = 1 THEN key := 'A'
                   ELSE BEGIN
                     exiting := TRUE;
                     key := 'E';
                   END;
      ELSE exiting := TRUE;                      { Pass back keystroke }
    END;
    popHilite (RunMenu, pick);                     { hilite new pick }

    IF key = 'A' THEN BEGIN                    { Do all demos on 'A' }
      ShowFile;
      TextChart;
      WalkPopup;
      Wait := ReadKey;                           { Wait for keypress }
      popErase (Walker);                        { Retreat thru popups }
      popErase (SalesChart);
      popErase (FileWindow);
    END;
  UNTIL exiting;
  popErase (RunMenu);
  RunResult := key;
END;

(* ---------------------------- *)

FUNCTION QuitResult : CHAR;

      { Pull down and act on Quit Menu }

VAR   key     : CHAR;
      pick    : INTEGER;
      exiting : BOOLEAN;

BEGIN
  pick := 1;
  exiting := FALSE;
  popShow (QuitMenu);
  popHilite (QuitMenu, 1);
  REPEAT
    key := Keystroke;
    popNormal (QuitMenu, pick);
    CASE key OF
      DownCursor : IF pick = 1 THEN pick := 2 ELSE pick := 1;
      UpCursor   : IF pick = 1 THEN pick := 2 ELSE pick := 1;
      'Q'        : exiting := TRUE;
      'E'        : exiting := TRUE;
      LeftCursor : exiting := TRUE;
      RiteCursor : exiting := TRUE;
      Enter      : BEGIN
                     IF pick = 1 THEN key := 'Q' ELSE key := 'E';
                     exiting := TRUE;
                   END;
      ELSE exiting := TRUE;                      { Pass back keystroke }
    END;
    popHilite (QuitMenu, pick);
  UNTIL exiting;
  popErase (QuitMenu);
  QuitResult := key;
END;

(* ---------------------------- *)

PROCEDURE DoMainMenu;

      { Manages pull-down menu selection }

TYPE MenuUp = (Demos, Run, Thru);

VAR  quitting : BOOLEAN;
     MMsel    : MenuUp;
     UserKey  : CHAR;

BEGIN
  Quitting := FALSE;
  MMsel    := Demos;
  REPEAT
    UserKey := chr (0);

    { Act on selected pulldown }
    CASE MMsel OF
      Demos : UserKey := DemoResult;
      Run   : UserKey := RunResult;
      Thru  : UserKey := QuitResult;
    END;

    { Act on returned keystroke }
    CASE UserKey OF
      'E'        : MMsel := Demos;
      'D'        : MMsel := Demos;
      'R'        : MMsel := Run;
      'T'        : MMsel := Thru;
      'Q'        : Quitting := TRUE;
      LeftCursor: IF MMsel = Demos THEN
                    MMsel := Thru
                  ELSE
                    Dec (MMsel);
      RiteCursor: IF MMsel = Thru THEN
                    MMsel := Demos
                  ELSE
                    Inc (MMsel);
    END;
  UNTIL quitting;
END;

(* ------------------------------------------------------------- *)

BEGIN  (*** Main program ***)

  { Initialize object text pointers }
  MainMenu.choice   := @MainMenuText;
  DemoMenu.contents := @DemoMenuText;
  RunMenu.contents  := @RunMenuText;
  QuitMenu.contents := @QuitMenuText;
  Walker.contents   := @WalkerText;

  { Set up screen and go }
  ClrScr;
  Cursoff;
  showMenubar (MainMenu);
  DoMainMenu;

  { Make sure cursor is on before quitting }
  Curson;
  ClrScr;
END.
```

End Listings

Shell Archives



by Allen Holub

This month's column looks a little like a feature of most of the Unix shells, the redirection mode (<<). A redirection command of the form program << pattern works as follows:

1. The command opens a temporary file.
2. The command copies standard input into the temporary file until it finds a line that exactly matches *pattern*. That is, all lines between the current one and the one that matches *pattern* are copied to the temporary file. The command expands shell variables in the copied lines unless *pattern* contains an escape character (a double or single quotation mark, or a backslash).
3. The command closes the temporary file.
4. The command redirects standard input to use the temporary file as its source. You can use this kind of redirection for two purposes. First, a shell script can create a second shell script on the fly, and then execute it. (Frankly, I'm not quite sure what this ability is good for; if you have a good application for a script creating a second script, send it to me and I'll print it.) The second purpose is to create the shell archive. A shell archive is a way of compacting an entire file system into a single file. Using a normal *mail* request, you can then send that file over a modem to another system. Once received, the other system can unpack the shell archive with a command like:

```
chmod +x archive
archive
```

or

```
csh <archive
```

Note that I've used the C shell here. In theory, the C, Bourne, and Korn shells should all work fine, but the C shell seems to be the most reliable. At least, archives sometimes don't expand when I use Berkeley's Bourne shell, and I'm not sure about the reason they don't. If anyone can enlighten me, please write.

The heart of the shell archive is the following command:

```
cat >file <<"end-of-document"
contents of file
"end-of-document"
```

The command *cat* either copies a list of files to standard output (like the MS-DOS type utility) or it copies standard input to standard output if no files are listed. I'm using the second behavior here. The simplest version of this program is in Example 1, below. You can speed up the version by using the unbuffered I/O functions and handling input in large chunks rather than one byte at a time. Here is another command:

```
cat >file <<"end-of-document"
contents of file
"end-of-document"
```

This command uses *cat* to copy all lines up to the one that starts with *pattern* into the indicated file. The command has *pattern* in quotation marks to prevent shell-variable expansion. A shell archive, then, is a series of these *cat* requests. You can also put other commands into the archive, as shown here:

```
mkdir 1 2
#
echo unarchiving 1/file1
cat >1/file1 <<"end-of-document"
contents of 1/file1
"end-of-document"
#
echo unarchiving 2/file2
cat >1/file2 <<"end-of-document"
contents of 2/file2
"end-of-document"
```

Two programs are presented this

```
#include <stdio.h>

main( argc, argv )
char    **argv;
{
    /* A simple form of cat */

    FILE *fp;
    int c;

    if( argc <= 1 )
        while( (c = getchar()) != EOF )
            putchar( c );
    else
    {
        for( --argc, ++argv; --argc >= 0 ; ++argv )

        {
            if( !(fp = fopen(*argv,"r")) )
                perror( *argv );
            else
                while( (c = getc(fp)) != EOF )
                    putchar( c );
        }
    }
}
```

Example 1: *The* cat *command copies standard input to standard output.*

month. The first program (*shellarc.c*) takes a list of file names and creates a shell archive. The program works in both the MS-DOS and the Unix system environments. The second program (*exp−arc.c*) is for MS-DOS systems that don't support << redirection. This program is a miniature shell that expands shell archives. You can use the two programs together which make it much easier to transfer blocks of related files either between Unix systems or between MS-DOS and Unix systems.

The program *shellarc.c* creates archives; it's in Listing One, page 86. You can use the program in one of two modes. The simplest way is just to list on the command line the files that you want to put into the archive. The program *shellarc.c* will create an archive from those files and send it to standard output. (You can redirect it to a file if you wish.) Note that the program retains any directory components of the path name. Here is an example: *shellarc george/washington*. This command will generate a *cat* command like this: *cat >george/washington <<pattern*.

It's up to you to create a *george* directory before you expand the archive. However, you can create the directory in the archive as well by inserting the commands necessary to do so, like this: *shellarc "=mkdir george" george/washington*.

All arguments that start with an equal sign are put into the archive word for word, but with the equal sign removed. If you're archiving many files, a better approach than using the command line is to use an archive-description file. The command-line syntax is as follows: *shellarc −ffile*, where *file* is the name of the description. The description file is line oriented. Each line that starts with an equal sign is written to the output verbatim. All other lines are assumed to be file names. The description file for the earlier example is as follows:

```
=mkdir george
george/washington
```

Two other command-line switches are recognized. A *−q* switch suppresses the quotation marks around the end−of−file pattern. If you use this switch, all shell variables that are mentioned in the source files are expanded. Normally, this behavior causes problems so the default is no expansion. A −. (minus, period) switch causes all file names to be preceded with a dot. Here is an example:

```
shellarc −. /foo/bar
```

This command generates the following:

```
cat >./foo/bar <<pattern
  . . .
pattern
```

The pattern used to mark the end of the file is defined on lines 52 and 53 of Listing One. The pattern should be something that's not likely to appear in an archived file; the default pattern is this:

```
END−OF−FILE− −END−OF−FILE − −
        END−OF−FILE
```

C CHEST

The quotation marks prevent the expansion of shell variables and are omitted if *−q* was specified. Most of the work is done in *main()* (lines 146–195). The *getargs()* call on line 157 parses the command-line arguments. The argument table is presented on lines 58–61. The program then loops (on line 169), getting file names from either the command line or the description file (*nextline()* handles both). The program opens each file, and then transfers it to standard output, surrounded by the appropriate *cat* request and end−of−file marker.

The second program is the dearchiver *exp−arc.c*, in Listing Two, page 101. As I said earlier, you don't need to use this program on a Unix system because the shell will do the expansion without difficulty. The program is intended for use on MS-DOS systems (or any other environment that doesn't support a << directive). The dearchiver is essentially a miniature shell that can do only two things: process << and execute commands. Imagine that the dearchiver sees input like this:

```
cat >foo <<pattern
  . . .
pattern
```

The dearchiver copies everything between *<<pattern* and the terminating pattern to a temporary file, redirects standard input to refer ence that file, and then executes *cat >foo* by using a *system()* directive. This behavior means that you'll need a *cat.exe* file somewhere on your system (just compile the earlier example if you need one) and that your normal shell must support normal output redirection. Normal usage is as follows: *exp−arc archive*.

Here *archive* is the name of the file. One command-line switch is recognized, *−e*, which causes the file's contents to be echoed to standard output as they're expanded. The *−e* is processed on lines 86–95 of *main()*. The loop that starts on line 100 gets *archive* names from the command line and processes them one at a time. On line 108, *extract_stuff()* processes one line of the input file. If that line has any other form than *cat >file <<pattern*, *extract_stuff()* returns *NOREDIR* and just executes the line (loaded into the array *command[]* by using *system()* on line 111. If the line does use that format *command[]* will hold the *cat >file* part of the line and word will hold the *pattern*. On line 114, *do_input()* is called to create (and initialize) the temporary file. The call returns the temporary file's name. On line 116, *freopen()* redirects standard input to reference the temporary file, rather than the console. On line 122, *system()* executes the *cat* and sends standard input back to the console

```
Old Style:

every(      good, boy, deserves, favor )
int         good;
long        boy;
char        *deserves;
double      favor;
{
  /* code */
}

New Style:

every(      int         good,
            long        boy,
            char        *deserves,
            double      favor
{
  /* code */
}
```

Figure 1: *C++-like subroutine declaration syntax*

C CHEST

on line 124. Note that if your shell doesn't support output redirection, you could use *freopen( )* to modify standard output in a similar manner.

Books

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Second Edition. Brian Kernighan and Dennis Ritchie's *The C Programming Language* was both the first—and for a long time the best—C textbook. Nonetheless, the language has evolved somewhat since 1978 and K&R was rapidly becoming obsolete. The situation has changed with the second edition of *The C Programming Language*. The authors have revised the book extensively, primarily to bring it in line with the Draft-Proposed ANSI standard, which is becoming stable enough to use as a model. I'm frankly dubious about some of the things that the authors have done in this regard. For example, the authors have wholeheartedly adopted the C++-like subroutine declaration syntax shown in Figure 1, page 118) without hinting about the portability problems that will result from use of this style. (In any event, I'm not sure whether the new style is a good idea—it's just a syntactic change and seems to do nothing except add additional confusion to the language, but be that as it may, K&R should have discussed the portability issue in greater depth.) Appendix A has been modified to be an adequate, though perhaps too concise, description of ANSI C, and there is a new addition, Appendix B, which discusses the ANSI I/O library.

The authors have wholeheartedly adopted the C++-like subroutine declaration syntax.

In other respects, the book has changed very little. It has both the same strengths and weaknesses as the previous edition. It's still a very good book if you're an experienced programmer who just wants a concise description of the language, without any fluff. The book is still one of the worst possible books for a beginning programmer or for a programmer who knows only one high-level language. The coverage of hard-to-understand parts of the language, like pointers, is still inadequate. (There's a new, but worthless, section called "Complicated Declarations" that, I suppose, is intended to rectify this situation somewhat. It doesn't.) There are other problems, too: the formatting style of the numerous examples is still not exemplary, there's too little white space, brackets are aligned inconsistently, and so forth. The variable names have improved somewhat from the previous edition—at least the variable names in the section that describes *alloc( )* have been improved—but there are still too many letters, i, j, and k, for my taste.

In summary, K&R is pretty much the same as before. On the other hand it's still a good language reference—though Harbison and Steele's *C, A Reference* (Prentice-Hall, 1987) is arguably a better one; and it's still a good technical introduction to the language—but only if you're a very experienced programmer. However, on the other hand, it's still a poor textbook if you're learning C as a first or second language.

Availability

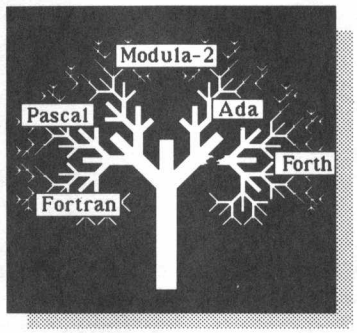
All source code for articles in this issue is available on a single disk. To order, send $14.95 to Dr. Dobb's Journal, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 88.)

Vote for your favorite feature/article.
Circle Reader Service Card **No. 6.**

Pop-Ups, Pull-Downs, and Menu Bars Revisited



by Kent Porter

In the November 1987 issue of *DDJ*, I did a feature article describing an architecture for pop-ups, pull-downs, and menu bars in Turbo C. Since then both Turbo C and Turbo Pascal have undergone major overhauls, particularly in the area of display management, and my architecture has changed accordingly. As a result, it's time to revisit the subject and give you an update, this time using Turbo Pascal 4.0.

Conceptually, the methodology borrows heavily from object-oriented programming. That is, it provides a couple of data structures for describing pop-ups and menu bars, and some procedures for manipulating them. The result, implemented in a Pascal unit, is a simple yet effective way of constructing friendly user interfaces. Just plug in the unit with a USES statement, then put the objects to work.

The productivity gains obtained from this architecture are by no means trivial. A few weeks ago, I wrote a complete stock market analysis program with five pull-down menus and who knows how many pop-up boxes in just three days of spare time. The visual prototype—all the menus and pop-ups and what not—took only a couple of hours. Packaged routines for the user interface left me free to concentrate on the internal workings of the program. This is important, since the user interface in PC applications typically demands more than 50 percent of the code. Anything that reduces this burden is by definition a productivity enhancement, so here goes.

The architecture rests on two basic data structures that I call popRec and menuRec, which are near the top of Listing One, page 103. They describe a pop-up and a menu bar, respectively. A certain synergy exists between the two, but we can treat them separately. Let's deal first with pop-ups.

There is no difference between a pull-down menu and a pop-up dialog or message box. A pull-down menu is simply a pop-up positioned so that it appears to hang down from the menu bar selection above. Both objects have fixed dimensions, borders, color attributes, and usually text contents. The only difference is the way you treat them—for example, moving a menu's highlight bar in response to keystrokes—which you can handle procedurally. Consequently, we'll refer to them from now on with the collective term *pop-up*, which effectively describes their nature.

A pop-up is a visual object that appears when needed and vanishes when it's no longer needed, restoring the area it overlaid during its brief existence. A screen write is destructive, so if you want to undo the damage it does, you have to save the image before flashing up the pop-up. Then, to make the pop-up disappear, you simply copy the saved image back into the video buffer.

This implies, of course, that you know where the video buffer is. POPUPS.PAS declares the pointer VideoBuffer. The initialization code (at the end of the unit) determines the memory address of display memory based on the current video mode and initializes the pointer. The unit assumes that the machine has an IBM-standard adapter (MDA, CGA, EGA, or VGA). You can make it more versatile by extending the possibilities to other adapters, such as the Hercules. The November 1987 issue of *DDJ*, pages 34–35 provide signatures for doing this.

Once you know where the video buffer is, it's a simple matter to save and restore it. To save: Allocate a heap node of the appropriate size and move that number of bytes from the video buffer to the heap node.

For text on an IBM-standard adapter, the display size is 4096 bytes. To make the pop-up disappear and restore the overlaid area: Move data back to the video buffer from the heap node and release the heap space.

The popShow and popErase procedures in POPUPS.PAS perform these operations automatically.

In a world characterized by pop-ups, you frequently operate within a hierarchy. For example, you pull down a menu, then pop up a dialog box, then display a message window. These objects must be removed in the proper sequence, which suggests a LIFO stack containing images. POPUPS.PAS doesn't provide a mechanism for managing the hierarchy; that's your responsibility. However, each popRec structure contains a placeholder—the save field—for storing the heap address of the previous image. PopShow loads this field when it saves the current image before altering the display. PopErase uses it to restore the old image, then releases the heap space and resets the pointer field to NIL.

Your responsibility, then, is to make sure that a sequence of popErase calls retreats in the opposite order from the series of popShow calls that created the present display configuration. This ensures that the display is not corrupted. Example:

```
popShow (pulldown_1);
  { Get a selection }
popShow (dialog_box_1);
  { Conduct dialog }
popShow (error_message_box);
```

```
    { Get a releasing keystroke, then
                                retreat }
popErase (error_message_box);
popErase (dialog_box_1);
popErase (pulldown_1);
```

The retreat thus occurs in LIFO order.

The popRec structure consists of three sections. The first ten fields are a descriptor list for the pop-up: its location/dimensions, border style, and color scheme. The next two fields are pointers to the optional fixed-text contents and the heap node. The final five fields are for state information saved by popShow and used by popErase to restore the display to its previous condition.

One of the key elements of this approach is, for each pop-up, to define a popRec structure as a typed constant. It's only necessary to initialize the descriptor list. Here's an example for an object called MyPop:

```
CONST MyPop : popRec = (
        left:10;
            { Location/dimensions }
        top : 5;
        right : 25;
        bottom : 10;
        style : 2;
            { Double-score border }
        normal : Cyan;
        hilite : LightCyan;
        normback : Blue;
        hiback : LightBlue;
        border : Black );
```

This pop-up appears between coordinates 10, 5 and 25, 10, has a double-score border, and uses the colors shown. The remaining fields are set by default to zero (NIL for pointers). All that remains to complete the description of the pop-up is to set the text contents pointer.

You can't initialize a pointer field at declaration, since addresses aren't known until run time. Therefore, if MyPop contains fixed text, you have to include a pointer assignment statement early in the program in order to complete the initialization of the structure. Say the fixed-text content is "This is MyPop." You can either assign the text to a variable string, or declare a string-typed constant, then put a pointer to it into MyPop. A typed constant is better because it conserves memory. Example:


```
CONST MyPopText : string[13] =
                        'This is MyPop';
BEGIN
   MyPop.contents := @MyPopText;
   . . .                { Do some stuff }
   popShow (MyPop);
                    { Display MyPop }
   . . .                { Do more stuff }
   popErase (MyPop);
            { Make MyPop go away }
   END.
```

When you execute popShow (MyPop), the pop-up appears with the text in the top row, with one blank between the border and the first character.

Now let's say MyPop is a pull-down menu with the selections Open, New, Close, and Quit, each on its own line. You can specify this list as follows:

```
CONST MyPopText : string[] =
   'Open~New~Close~Quit';
```

The tilde character (~) is the element separator, defined as the constant SEP in POPUPS.PAS. The popWrite procedure (nested within popShow and thus not externally callable) moves to the second column of the next row when it encounters a tilde within a string.

You can also use Writeln( ) statements, as well as cursor control statements such as Gotoxy, to output to the active pop-up. (The "active pop-up" is the one most recently placed on the display.) The region inside the border is a clipped window whose upper-left corner is at coordinates 1, 1. Its physical dimensions are (right-left-1) columns and (bottom-top-1) rows. For example, if top = 10 and bottom = 20, then the window has nine rows numbered 1..9. While a pop-up is active, screen writes remain within its borders, wrapping at the right edge and scrolling the contents at the bottom.

The unit also furnishes popCenter, which writes a string that's centered on the indicated row. This routine verifies that the target pop-up is currently visible and that the row is within the window; it doesn't verify that the target is active. For this reason, it's important that you not write to a visible but inactive pop-up using popCenter; you'll corrupt the active object, which owns the current text window.

PopShow calls the Textbox procedure to outline the pop-up based on the object's style and border fields. The border field specifies the text foreground color for the outline. Style specifies one of three border types:

```
0 = No border
1 = Single-score
2 = Double-score
```

Textbox uses this value, which it receives as a parameter, to select the appropriate box-drawing character set from the typed constant array called Border. If style is 0, the routine merely returns without taking any action.

PopShow saves the current video state (window dimensions, foreground/background color, and cursor position), resets the window to the full screen and saves the display image, and then draws the border for the pop-up using the object descriptor list. It then sets the window to the area inside the border and issues a ClrScr to implement the new background color and get rid of any overlaid text. Finally, it calls popWrite to place any fixed text within the pop-up. PopErase reverses this process by restoring the previous screen image and its state, including the previously active window. This explains why you must enforce LIFO order in removing multiple pop-ups.

PopHilite and popNormal change the color scheme of a given row within the active pop-up. Initially a pop-up appears with the normal foreground/background colors specified in its descriptor list (the normal and normback fields). PopHilite changes the entire row to the highlight colors (hilite and hiback), and popNormal changes it back. The real work of these routines is done by the shared *procedure popRewrite*, which uses ROM BIOS calls to read each character in the row and write it back to the same location with the new color attributes.

PopHilite is useful any time for emphasizing fixed or variable text within a pop-up. The combination

of popHilite and popNormal affect a movable lighted bar that indicates the current pull-down menu selection. Consider for example Example 1, this page.

In other words, when the down-cursor key is pressed, you first restore the selected row to normal, then move to the next row and highlight it. Other keys—up-cursor, home, end, and so on—trigger similar sequences. This logic is illustrative and not complete; you also have to handle wrapping, as when the selected row is at the bottom of the menu and the user hits down-cursor. A sample program presented later illustrates the logic more fully.

But first let's talk about menu bars. As implemented here, a menu bar is a distinctively colored list of selections arrayed across the screen or accross a pop-up, usually, but not always, at the top. The top row of the Turbo Pascal environment is a typical example. When you select something from the menu bar, a pull-down menu might appear beneath it, allowing subselections in a tree-like fashion. Or a menu bar selection might be an "end node" such as the Run selection in the Turbo Pascal environment; it doesn't produce a pull-down menu, but instead initiates an action directly.

Like pop-ups, menu bars can be described and initialized with a data structure. In POPUPS.PAS, the structure is called menuRec. Its components are:

Row—The row (1..n) where the menu bar will appear within the window that contains it.
Interval—The spacing between first characters of the menu choices.
Fore—The text foreground color.
Back—The text background color.
Choice—A pointer to the string containing the menu choices.

It's necessary to calculate the interval manually, taking into account the number of items on the menu and the width of the containing window. The latter is usually the whole screen but it might be a pop-up that has its own menu bar. The formula is Interval = (W DIV I) + 1, where W is the total width and I is the number of items. Let's say you're planning to place a menu bar with

```
SelectedRow := 1;
popHilite (MyPop, SelectedRow);
REPEAT { Menu selection process }
   UserKey := Keystroke;
   IF UserKey = DownCursor THEN BEGIN
        popNormal (MyPop, SelectedRow);
        INC (SelectedRow);
        popHilite (MyPop, SelectedRow);
      END
   ELSE
      . . .
UNTIL TerminatingCondition;
```

Example 1: *PopHilite is useful for emphasizing fixed or variable text within a pop-up.*

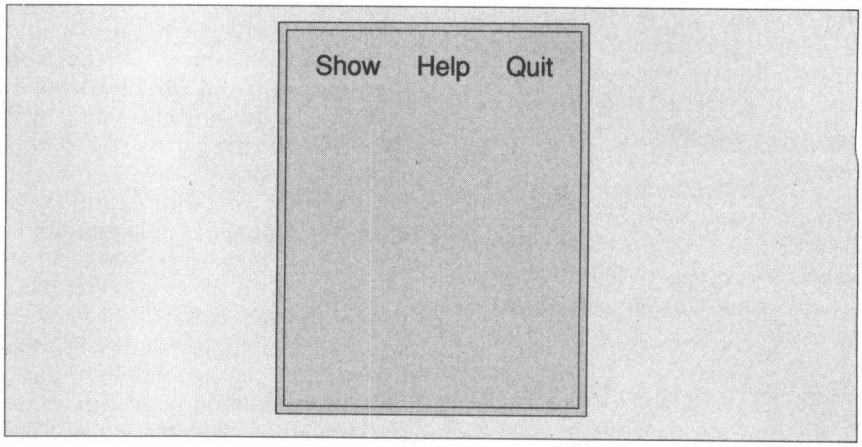


Figure 1: *Screen image after the popShow and showMenubar statements have been executed.*

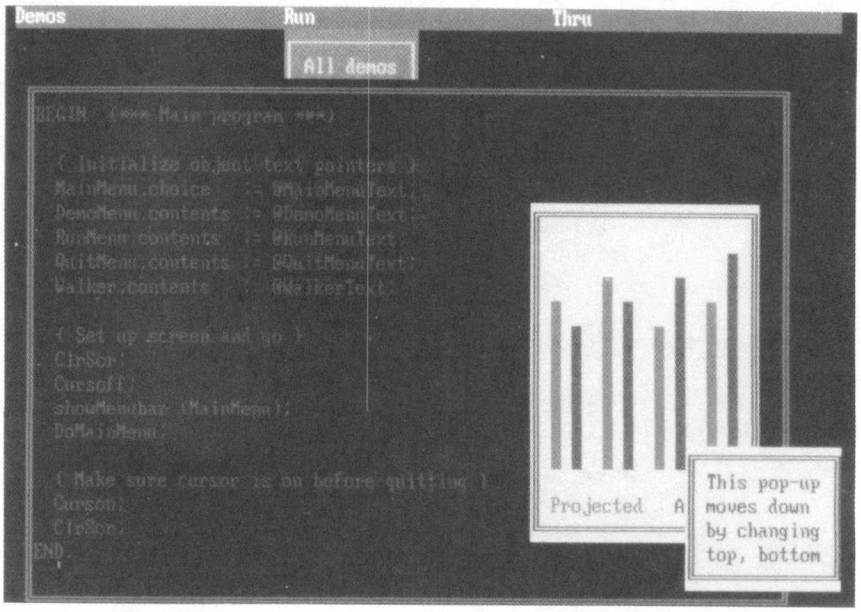


Figure 2: *This snapshot shows most of the visual objects defined in POPDEMO.PAS. Note that the objects can overlay each other, and when they go away, the overlaid surface is automatically restored. Defining such visual environments is fairly simple using the tools in the POPUPS.PAS unit.*

(continued from page 125)

three selections in a pop-up called YourPop. YourPop begins at column 30 and ends at column 47. The width of the window, then, is 47 − 30 − 1 = 16. Therefore, you can estimate the spacing as Interval = (16 DIV 3) + 1 = 6.

Note the word *estimate*. The calculated interval might not work if the last selection is a long word. It takes some trial and error; you must adjust the interval to make the menu text fit the space without wrapping to the next row.

As in the case of a pop-up, you must specify the menu bar string separately. The same element separator applies. Say the menu bar contains the choices Show, Help, and Quit. You can define it as

```
CONST YourPopMenubar: string[14] =
   'Show ~ Help ~ Quit';
```

Similarly, you can define the menu bar structure as a typed constant. Example:

```
CONST YourPopMenu : menuRec = (
  Row     : 1;
  Interval : 6;
  Fore    : Black;
  Back    : LightGray);
```

Then, at run time (assuming YourPop has also been defined), do the following:

```
BEGIN
   YourPopMenu.choice =
                  @YourPopMenubar;
   ...                { Do some stuff }
   popshow (YourPop);

   showMenubar (YourPopMenu);

   ...          { Act on the menu bar }
popErase (YourPop);

...                          { Do whatever }
END.
```

After the popShow and showMenubar statements execute, you have an object on the screen that looks something like Figure 1 (see page 125).

The synergy between menu bars and pull-downs occurs with respect to the interval. Define your pull-downs so that they hang under the related menu bar selections.

Again, this is a manual process in this architecture, and you might have to do some tweaking to get it right. A case in point is where the last menu bar selection is at column 70, but you need 12 columns for the pull-down. Obviously, you can't go to column 82 on an 80-column display, so you'll have to shift the last pop-up to the left.

The POPUPS.PAS unit also provides three other services commonly needed in interactive environments. Two are related: Curson and Cursoff. They turn the text cursor on and off, respectively. The Keystroke function returns a value derived from the user's operation of the keyboard.

Keystroke plays some games with values for the special extended keys. These encompass F1–F10, the cursor control and Home/End/PgUp/PgDn keys, and any Alt-key combination. All of these generate a 2-byte sequence, with the first being ASCII 0 and the second denoting which key was actually pressed. The problem is that the second byte might be a valid alphanumeric. For example, the Home key generates 0 and 71. It happens that 71 is also the ASCII value for uppercase G. If Keystroke returned only the second byte, your program would be unable to determine if the user pressed G or Home.

For this reason, Keystroke adds 128 to the second byte of any extended key and returns that value. The unit defines names for some of these keystroke values. You can easily expand the list for others. For example, F2–F10 return second bytes of 60 . . 68, which Keystroke maps to 188 . . 196.

The program POPDEMO.PAS in Listing Two, page 110, is a fairly extensive demonstration of the user interface architecture. This program runs three tasks in different pop-ups:

- Lists its own source
- Walks a pop-up down the right side of the screen
- Produces simple column chart using text graphics

It has a menu bar and three pull-down menus. Navigation through the menu structure is accomplished by any combination of cursor keys, initial letters, and/or the Enter key. For example, if you're in the Demos menu and press either the left cursor key or T, you'll instantly move to the Thru menu. To exit the program, press either Enter or Q. If you press E from any menu, you'll return to the Demos menu. In Demos, if you want to see the business chart, press G or cursor to it and hit Enter. Figure 1 shows what you get on a color monitor if you take the All demos selection from the Run menu.

Most of the program's work is done in two places: the object initialization at the start and the DoMainMenu procedure. The initialization defines all the visual objects of the program. DoMainMenu acts on keystrokes and dispatches the appropriate pull-down. Note that each pull-down returns its terminating keystroke. DoMainMenu then uses this result to decide what to do next.

The architecture in POPUPS.PAS removes virtually all of the complexity in managing pop-ups, pull-downs, and menu bars. The only remaining complexity is in handling signals from the user; interpret numerous possibilities and act on them appropriately. this still requires consider1able code. Unfortunately, this is too application-dependent to be easily reduced to abstract, general-purpose code.

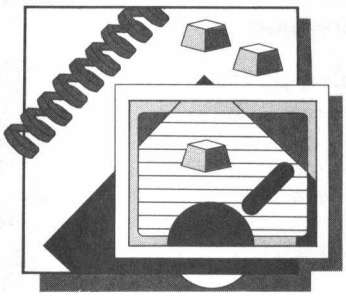
Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

(Listings begin on page 103.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

A Case Study in Paradigm Clash



"IBM Neural Network Chips Make Everything Obsolete" blared the headline. In the April 4, 1988, *InfoWorld*, columnist John Gantz blew the lid off IBM's new strategy for eliminating the need for programmers with trainable neural network computers.

Of course, it was an April Fools' gag. You just knew it had to be, even though it was three days late for April Fools', because Gantz is too skeptical to fall for such wild claims. Sort of like *DDJ* readers. When it comes to choosing between blue-sky prognostication and solidly grounded technique, *DDJ* readers typically put ground over sky.

So, why (in an informal survey I conducted last year) was parallel processing the topic most readers wanted to see more of in the magazine? Sort of blue-sky, isn't it? This was hardly a topic of great immediate practical value to a professional programmer, considering how few parallel architecture machines there are.

The trick, I think, is the word "immediate." After all, to a long-distance runner, the only activity of immediate practical value is performing well in races. But no runner spends as much time competing as he or she does practicing and working out. I try to address that need in this column: to point out the new exercise equipment, and (when I can) to give some suggestions for its use.

In the first two installments, I touched on (superficially) several programming paradigms: object-oriented programming, logic programming, and communicating sequential processes. Although the intent here is to explore new intellectual tools rather than to develop mastery of familiar ones, I hope to get deeper into each of these and other paradigms in subsequent columns.

by Michael Swaine

This month I'm going to show an algorithm for a new paradigm of programming. But first, I have to talk about current research in cognitive psychology.

But Why, Mike?

Well, first, to show what's entailed in a clash of paradigms. I did graduate work in cognitive psychology a few years ago, and since that time, a new paradigm has come into prominence in that field. This new paradigm challenges some operating assumptions of the field, assumptions that I took seriously back then. I believe that, in the small resolution that is going on today in cognitive psychology, there is a useful example of what happens when paradigms collide—useful because there are programming paradigms on a collision course today.

Second, to give an application-oriented perspective on the aforementioned algorithm. You see, the new psychological paradigm is also the new programming paradigm—um, neural networks. As it happens, from a paradigmatic point of view, Gantz's joke may not be so funny after all.

What's This Paradigm Stuff?

Before the psychology, a reminder is in order about paradigms and why we should care about them. In 1962, philosopher of science Thomas S. Kuhn published a book titled *The Structure of Scientific Revolutions*, which enjoyed a great vogue in the late sixties and early seventies in undergraduate and graduate curricula. *DDJ* associate editor Ron Copeland says he read Kuhn for four different classes. I think I only read him three times, but I read him carefully.

Kuhn advanced the then-controversial thesis that the kind of science a community of scientists will produce depends on their shared values, terminology, techniques, model problems, and concrete examples of how to solve such problems. He subsumed these things under the term "paradigm," and described how disorienting it was to move from one paradigm to another, and how difficult it was for those working within one paradigm to communicate with those working in another.

With the pain came gain. Kuhn said, "Paradigm shifts—troubled periods in which basic assumptions of a discipline are being upset—are one of the means by which science progresses."

It was Kuhn's insight to apply this concept of paradigms to the growth of science. Its applicability to an engineering discipline such as software development is more obvious. It's not really controversial to note that those working with different tools and assumptions and languages will see different problems to solve and will create different kinds of artifacts. It's not controversial, but it is important.

The differences will be no less real if they are invisible to the user. If one programmer cooks a spaghetti-code general-ledger program in Basic and another programmer models the elements of a pre-existing paper general ledger in Smalltalk, the delivered goods may look identical to the user. (Not likely, but it is possible.) But the program's result will, in fact, be very different things, as will be evident to other programmers hired to maintain them.

The fact that the program that you create depends on the paradigm within which you create it should matter to you. I'm sure it does. But it also implies that you should know what paradigms are available, understand them well enough to know what problems they solve, and have the flexibility to move from paradigm to paradigm at will.

Another reason to understand other paradigms is communication. As I have mentioned before, I think that we are seeing a paradigmatic broadening of the discipline of programming. If programmers continue to be educated (and to educate themselves) within narrow paradigm boundaries, it will become increasingly difficult for programmers to learn from one another.

Paradigm differences run deeper than just different programming languages and algorithms. Learning object-oriented programming, for example, is not just a matter of picking up some new techniques. If you've spent your professional life thinking that programming is really a matter of finding the right algorithm and implementing it efficiently, object-oriented programming will seriously warp your thinking.

This is something similar to what some psychological researchers are facing today.

A Crash Course in Cognition

The text for this month's exploration into psychology is *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volumes 1 and 2*, by David E. Rumelhart, James L. McClelland, and the PDP Research Group (A Bradford Book; MIT Press, 1986). I'll synopsize it, along with the mainstream of cognitive psychology against which it flows.

Rumelhart, et al., present a class of models (called parallel distributed processing or PDP, models) in which the mechanism of information processing is assumed to be the interaction of large numbers of simple processing elements, each sending excitatory and inhibitory signals to other units. The units may correspond to various mental entities (hypotheses, goals, or potential actions) or aspects or features of such entities.

PDP is more or less equivalent to neural networks. The models the authors subsume under the name PDP all have the following:

- A set of processing elements
- A pattern of connectivity of the elements
- A state or level of activation for each element
- An activation rule that combines inputs to an element to produce a new activation level
- An output function that produces an output from an element based on its activation level
- A learning mechanism that changes the pattern of connectivity on the basis of "experience."

This is really a model of the network of real neurons that make up the brain, although the neurons of the models are highly idealized (see Figure 1, page 132). One of the important ways in which the individual models differ is in the mechanism

(continued from page 131)

of learning. But they do agree in this: learning in a PDP model is a purely local phenomenon. No "supervisor" rewards the elements for good choices. All levels of learning are built from simple, strictly local feedback mechanisms.

Levels of Processing

One problem that any theory of cognitive processes must face is the apparently vast gap of levels of processing from the cognitive down to the neurological.

On the one hand, it doesn't make much sense to develop models of thought that are in conflict with what little we know about the functioning of neurons in the central nervous system.

But on the other hand, to assume that the same principles of organization that explain the interaction of neurons will also explain how thoughts interact when we read a novel seems like wishful thinking (like imagining that colleges could replace chemistry courses with courses in physics). Chemists may accept that their science is in some sense implicit in the science of physics, but they still feel that chemical problems require chemical solutions.

No research psychologist today rejects the reductionist principle that complex cognitive behavior is based on the functioning of neurons in the central nervous system. But in the 1970s, many scientists began to question whether there was any practical significance in that principle. Cognitive psychology in the past two decades was increasingly likely to seek purely cognitive models for cognitive processes.

This is tough. It's tough to bootstrap a science. As a graduate student, I designed an experiment to distinguish between storage and retrieval techniques in the comprehension of structured information. The experiment was cleanly designed, I think. Because of necessary temporal sequences, its results should have clearly established whether certain things were happening during the storage or the retrieval of the information from memory.

Fortunately others had better luck than I with cognitive psychology.

What my experiment lacked (I discovered too late) was a solid theoretical foundation. Yes, it distinguished some sort of storage effect from some sort of retrieval effect, but without some accepted model of what was stored and retrieved, I couldn't communicate any useful information to other researchers. There were models in the light of which I could have interpreted my results. Since I had not designed my experiment with any of those models in mind, my results were hard to compare or connect with the results that those models produced. I was in a giddy state of "paradigmlessness." The words "skew" and "incommensurable" come to mind.

Fortunately others had better luck than I with cognitive psychology. But my experience is suggestive of what some researchers may be feeling today.

The Agony of Incommensurability

The PDP model does assume the same principles of organization that explain the interaction of neurons will also explain how thoughts interact when we read a novel. In this way, it is in conflict with the cognitive psychology I learned. But conflict is the wrong word. Some of the kinds of experiments that the PDP researchers are doing produce results that are incommensurable with other results, in several ways:

1. Although the book identifies many high-level mental phenomena as appropriate for PDP modeling, conclusive research results in PDP research are likely to come at the lower levels of processing. If clusters of neurons can be shown to act in accord with the models, the research will provide a base of accepted results on which to build experiments addressing higher levels of processing. Until then, the cleanest results in PDP research are likely to be at a level incommensurable with results in more traditional cognitive research. PDP research may produce excellent oranges to compare with the apples of cognitive psychology.

2. Many of the studies cited in the book are computer simulations. The founders of PDP models did not introduce simulation as a psychological research tool, but they do have to deal with the problems of the approach. A simulation is a theory; it makes predictions that are

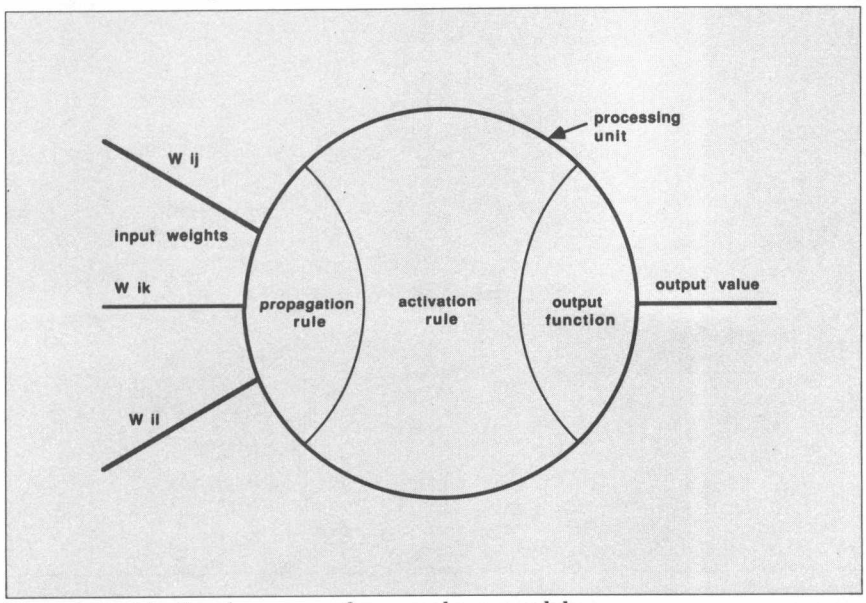


Figure 1: *Idealized neuron of a neural net model*

confirmed or refuted by examining the real-world process being simulated. Ideally, if the simulation doesn't match (on the appropriate measures) what it is supposed to simulate, the theory should be rejected. In practice, when a new paradigm is being explored, a faulty simulation will just get tweaked until it works. In the case of PDP models, you could tweak the following:

- The pattern of connectivity
- The initial state of activation
- The activation rule
- The output function
- The learning mechanism

With so much freedom, you could make any simulation work. PDP researchers will have to nail down which of the variables in the simulation are free parameters and which are fixed. Essentially, they have to sort out the neural software from the firmware. Only then will they be able to build testable theories. It is typical in the development of a theory to try various versions to see which is fruitful. But this makes PDP (like a political candidate who changes his positions on the issues) a moving target for alternative candidates.

3. Finally, if PDP work proves fruitful, and if I am right about the need to work upward from low-level processes in PDP research, traditional researchers may have to abandon their accustomed research techniques, tools, and subject matter. A psychologist who had been studying attention and perception at a fairly high level could find himself focusing his career in on saccadic eye movements. He might not find this a comfortable move.

All of this makes it hard for those working within a different paradigm even to talk with PDP researchers.

But the Title of the Column Is. . .

. . . Programming Paradigms, right. So how does all this tie into programming? Aside from the putative benefit of understanding what paradigm clashes mean in a different context, do neural networks have any real interest for programmers? I think they do, and I think they may one day raise some disturbing questions about just what it means to be a programmer.

Many people are now beginning to take neural networks seriously as a programming technique. A 1986 conference sponsored by the American Institute of Physics in Snowbird, Utah, drew 160 people. A friend told me while I was writing this column that his chief programmer had just quit to develop neural networks.

What Are Neural Nets Good For?

What kinds of problems can and can't be solved by neural network techniques? That's pretty straightforward. Neural networks can solve any conventional computational problem. The proof of this is also straightforward. First, note that a computational problem can be represented by a set of Boolean functions. Second, recall that any Boolean function can be built entirely from two-input NAND gates. If a NAND gate can be simulated by a component of a neural network, it follows that

neural networks are formally capable of solving any problem solvable by computer.

This happens to be the case. Figure 2, on page 134, shows how to model a NAND gate in a neural network.

All right, but what kinds of problems can a neural network approach solve efficiently?

The first limitation is an absolute one. Neural networks are instances of parallel processing. They can be expected to produce benefits in those areas in which parallel processing is potentially beneficial, but they can produce no gain where parallelism is not beneficial. If processing power is more precious than time, a sequential solution is the right solution.

When time is a factor, the best that a neural network (or any parallel approach) can do is to reduce processing time by a factor equal to the number of processors. For structured problems (i.e., problems with relatively short algorithms) Abu-Mostafa argues that the efficiency of neural networks is likely to be much less than this. For problems requiring long algorithms (what are called random problems) the efficiency may be reasonable (i.e., polynomial in the number of processors).

A tentative conclusion is that neural networks are more useful for large, random problems.

This is supported by the fact that neural networks are built of very simple processing units, so the processors should ultimately be cheap and plentiful. They could be particularly cheap and plentiful if imple-

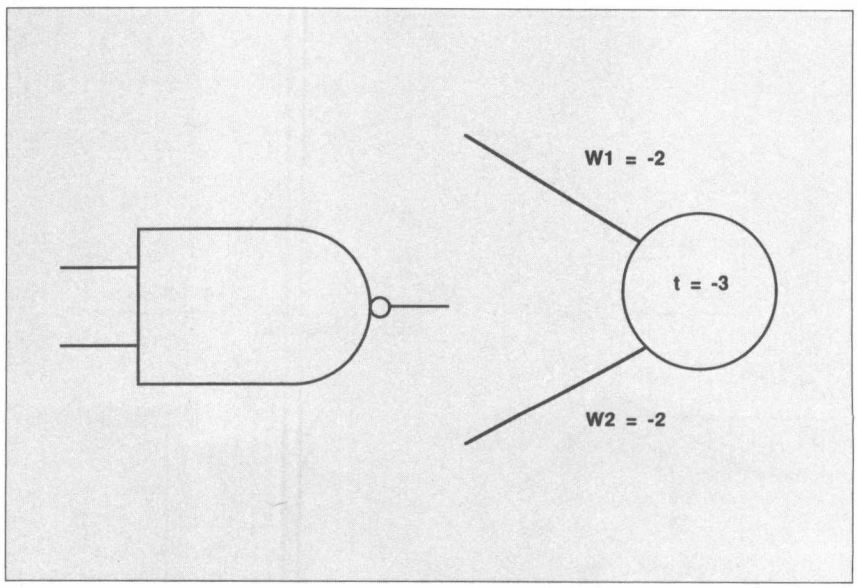


Figure 2: *A NAND gate can be simulated by a neural network node with a threshold −3 and two inputs of weight −2.*

mented in optical-device technology, for which neural networks look like an ideal candidate. The size-of-problem issue gets more definite when you consider the algorithm embedded in a neural network solution. Abu-Mostafa says that the time complexity of the problem is accommodated by the number of steps, the space complexity by the number of processing units, and the Kolmogorov complexity (or complexity of algorithm) is accommodated by the degrees of freedom (or information capacity) of the synaptic connections. These measures of complexity are not independent, and, in fact, the Kolmogorov complexity will be very large if the space complexity is large.

Thus, a problem that is demanding in terms of space complexity, but modest in terms of Kolmogorov complexity, will waste a great deal of information capacity. Exponential-time problems (like the Traveling Salesman Problem) similarly waste capacity in neural nets.

What this means is that problems requiring a lot of computation time or memory, but having simple algorithms, will use neural networks very inefficiently. Problems that require very large algorithms will make better use of neural networks. Pattern recognition in natural environments is one example of the latter kind of problem.

Programming Paradigms Clash

Another area in which neural networks could be useful is in the unsupervised search for solutions to open problems. The hope of neural network research in psychology is that very complex and high-level processes (i.e., complex algorithms) can be built up from simple mechanisms without direction or supervision.

In psychology, this amounts to a very strong claim about the nature of human learning. In programming, it brings us to the edge of a potential paradigm clash.

To many of us, computer programming has long meant selecting or developing the algorithm or algorithms to solve the problem. The object-oriented programming and declarative programming paradigms challenge that view by putting the emphasis on modeling the problem accurately. But how would we feel if we only had to state the problem and wait while the system developed an algorithm? And what if we couldn't understand the algorithm?

Remember the reaction among mathematicians to the unorthodox computer-assisted solution to the four-color theorem? Neural network technology will never make programmers obsolete, but it may one day present them with an identity crisis. There's one elementary point about the efficiency of neural networks that I haven't stated explicitly: neural network techniques require parallel architectures. There are few computers capable of supporting parallel processing today, and neural net algorithms like that presented here are of absolutely no practical value without true parallelism.

Well, there is one "practical" value. They're good workout room equipment for learning about new paradigms and flexing your intellectual muscles.

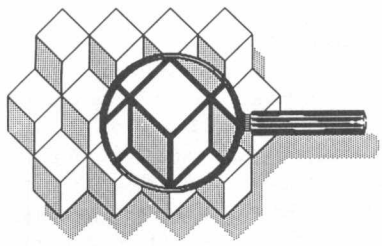
Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for you favorite feature/article.
Circle Reader Service **No. 9**

OF INTEREST



Tools for Programmers

FairCom has announced d-tree, a set of applications generation modules which can be used as is, or can be modified to suit a programmer's specific needs. A comprehensive set of application development aids is supplied: data dictionary, program dictionary, file reorganization utility, and program generator. d-tree provides routines to extend the toolboxes included with c-tree (FairCom's file handler) and r-tree (FairCom's report generator). d-tree supports all commercial grade C compilers. It requires 256K memory. The introductory price is set at $495. Reader Service No. 20.

FairCom
4006 W. Broadway
Columbia, MO 65203
314-445-6833

Big Bang Software has released an MC68020 symbolic simulator/debugger for the professional cross-developer. The software package, a companion to the MC68000/10 simulator released in 1986, enables the user to test and debug 68020 software on IBM PCs or compatibles. Hex files of Motorola S-Records can be disassembled and displayed. Instructions can be executed in either fast or single-step mode. Results of each executed instruction on registers, flags, and 68020 memory are immediately available for display. During single-stepping the display shows the last instruction executed, the current contents of all registers, and the instruction following the last executed. All 68020 instructions, addressing modes and condition codes are fully supported. Load, dump, and breakpoint facilities are included. The product runs on IBM PCs, IBM PC XTs, IBM PC ATs, and compatibles with DOS 2.0 or later and 256K. The price is $395. Reader Service No. 21.

Big Bang Software
Beachwalk Centre
7151 W. Hwy. 98, Ste. 286
Panama City Beach, FL 32407
904-784-7114

Fastmath is a math programming language which is available from **21st Century Data**. Fastmath records and runs a program as the programmer enters it. Users can modify command entries using the built-in editor and execute completed programs nonstop using run mode. Fastmath performs arithmetic, algebraic, logical, and advanced mathematical point display. Other arithmetic modes include signed and unsigned decimal, hexadecimal, octal, and binary. The program conforms to the specifications of IEEE Binary Floating Point Arithmetic Standard Number 754. All programs created with Fastmath are fully portable. The product is available for the Macintosh, IBM PCs, IBM PS/2s, and compatibles. Reader Service No. 22.

21st Century Data
P.O. Box 1139
Solana Beach, CA 92075
619-755-6218

Genus Microprogramming has announced the PCX Programmer's Toolkit, which provides complete documentation and software that allows software developers to create applications with the ability to display, save, capture, and manipulate PCX format images. The Toolkit supports all display modes of the Hercules, CGA, EGA, and VGA display adapters, and contains a comprehensive set of compiler interfaces. These interfaces include Microsoft C, Quick C, Turbo C, Lattice C, Quick Basic, Turbo Basic, Turbo Pascal, and Clipper. Except for Turbo Basic, libraries are provided that are directly linked to the user's program, and Quick Libraries are provided for the Microsoft integrated compilers. All routines are written in assembly language.

Additional features include several utility programs that display and capture screens, cut out sections of an image, locate screen coordinates, inspect image headers, and manage image libraries. An image library manager allows images to be grouped together and compressed, saving as much as 50 percent of disk space. The Toolkit is priced at $89.95. Source code is available for an additional $100. Reader Service No. 23.

Genus Microprogramming
6305 Mobud Dr.
Houston, TX 77074
713-771-4914

EPEC, from **Jandi Technologies**, integrates a traditional text editor and a basic word processor together under a proprietary windowing system that supports true windows and true multitasking. Some of the features of EPEC include: a windowing environment; cross window editing; joint window editing; multiple undo's and redo's; range copying and deleting; area cutting, pasting, editing, and text reformatting; separating print attributes from the main text; extended ASCII graphics; cross reference generations; and key stream assignment to a function key or file. EPEC is available for IBM PCs, IBM PC XTs, IBM PC ATs, and compatibles. The package comes with keyboard-key templates and a 200+ page user's manual and is priced at $99. Reader Service No. 24.

Jandi Technologies
155-U New Boston St.
Woburn, MA 01801
617-932-0629

C Products

The C Programmer's Association has been formed to provide C programmers with inexpensive C functions and to offer C programmers the opportunity to make profit from the C functions they have written. Programmers can purchase single C functions or entire libraries. Programs can be ordered through the C Programmer's Association's bulletin board. All functions purchased are

licensed to members only. Members may include these functions in programs intended for commercial distribution.

Programmers may also submit C functions to the association. If the function is accepted by the association, the programmer will receive a percentage of each one that is sold. Reader Service No. 25.
C Programmer's Association
10668 Ellen St., Ste. G
El Monte, CA 91731
818-442-1522

Whitesmiths Ltd. is now shipping its new ANSI standard Version 3.3 optimizing C cross compiler for the Motorola MC68HC11 and new interactive C source-level cross debugger, CXDB/6811. The packages run under PC- and MS-DOS, VAX/VMS, and on the Sun and Apollo workstations. The features of the cross compiler include support for five programming models, a fast function calling mechanism, and support for C source-level debugging. CXDB/6811 is an interactive C source-level debugger which is fully integrated with Whitesmiths' cross compiler. The program allows users to debug target code in terms of original C source code using the resources provided by the host system and is used as a first-phase or logic debugger.

License fees for the C cross compiler range from $1,500 to $7,000. License fees for the C cross compiler and CXDB/6811 together range from $2,500 to $12,000. Reader Service No. 26.
Whitesmiths Ltd.
59 Power Rd.
Westford, MA 01886
617-692-7800

C_talk from **CNS Inc.** is an object-oriented programming environment for the C programming language. It adds to C the essential ingredients of object-oriented languages, namely: encapsulation, inheritance, and a dynamically-bound messaging mechanism. C_talk comes with a set of software tools to help programmers get started with object-oriented software engineering: a powerful Smalltalk-like Browser for defining and editing classes of objects; a preprocessor to convert C_talk object class descriptions into standardized C source files that are compatible with the most popular C compilers; a semiautomatic *make* utility for controlling the preprocessing, compiling, and linking of an application program; and a set of predefined objects, called foundation classes, to use as basic building blocks in getting an application started.

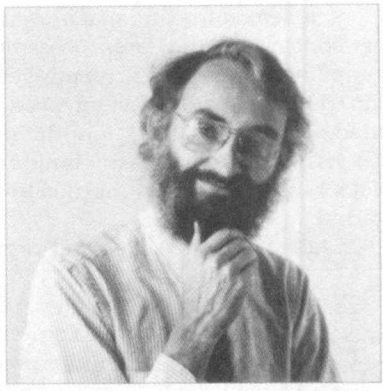
C_talk runs on IBM PCs and compatibles with 512K. It works with Microsoft C, Turbo C, Lattice C, and Computer Innovations C86. A hard disk and a mouse are recommended. The product sells for $149.95. Reader Service No. 27.
CNS Inc.
7090 Shady Oak Rd.
Eden Prairie, MN 55344
612-944-0170

Micro Computer Control (MCC) has announced CLIB-521, a C language support library for the Advanced Micro Devices 80C521 single-chip microcontroller, compatible with MCC's MICRO/C-51 cross-compiler. CLIB-521 provides fully developed and tested MICRO/C functions for chip-specific peripherals. Included in the package are functions that perform fast block moves, maintain an external RAM pseudo stack, initialize and control the Watch-Dog Timer, and control entry and exit from Idle and Power Down Modes. Also provided is an automatic memory status check to help recover system operation after a power-down reset. CLIB-521 includes the CLIB linkable library file, source files for all library functions, and 80C521 Processor Descriptor File which provides direct access to all on-chip peripherals directly from the MICRO/C language, and a user's guide including application examples. Price for the package is $125. Reader Service No. 28.
Micro Computer Control
P.O. Box 275
Hopewell, NJ 08525
609-466-1751

Migent is now shipping The Developer's Toolkit for C Language, which is a complete set of C libraries. The Toolkit also contains a personal engine, which can be resold without royalties, report writer, forms generator, database administrator, and import/export capability. All resulting applications immediately become multiuser when placed on a LAN with the Emerald Bay Database Server. Suggested retail price is $495. Reader Service No. 29.
Migent Inc.
865 Tahoe Blvd.
Incline Village, NV 89450
702-832-3700

DDJ

SWAINE'S FLAMES



P.J. Plauger, contributing editor
Computer Language Magazine
P.J.,
I heartily endorse your argument, in the April 1988 issue of *Computer Language*, for not crediting the authors of your ideas. I like the way you extended the principle beyond mere programming by snubbing Anton Chekhov in that same issue. Sure, Isaac Newton comes off as magnanimous in that famous quote about seeing farther than other men because he stood on the shoulders of giants, but I don't recall that he identified the giants. Could Newton have continued to see farther than others if he had told them where he was looking? I think not.

Besides, references are ugly. When HyperText becomes a reality, every document will include its own intellectual audit trail. Until then let's not clutter up the pages of magazines with boring academic citations. Why would anyone want to read further about a topic discussed in *Computer Language* anyway?

What I meant by that, of course, was that *Computer Language* always publishes the last word on any topic.

So, thanks for the idea, which I plan to use. Don't worry, though; whenever I do, I'll cite what I call the Plauger Principle.

Magnanimously,
Corbett

Robert Cringely, field editor
InfoWorld
Bob,
I think you missed a bet in your instant analysis of John Sculley's comments on suing Microsoft. What Sculley said in *Odyssey* was, "If we sued our most important software supplier, our business customers would think we'd lost our minds." Clearly, Microsoft is no longer Apple's most important software supplier. Do I get a T-shirt or something for this?

Expectantly,
Corbett

David Bunnell, editor-in-chief
Altair Computer Notes, etc.
David,
The legal ramifications of publishing are getting more and more complicated. Thanks for telling us about the important Congressional report, Science, Technology, and the First Amendment. I am writing to see if I got the facts straight, since I forgot in which of your magazines you discussed it. As I recall, the report costs $3.50, its GPO stock number is 052-003-01090-9, and I can get it by writing to the Superintendent of Documents, Government Printing Office, Washington, DC 20402-9325. Is that right? By the way, how's the *Windows* magazine coming along?

Curiously,
Corbett

Susan Gubernat, editor
Publish! (sic) Magazine
Susan,
I'm sorry that you were unable to use the copy of *Microsoft Systems Journal* that I sent you for your annual redesign issue. At least you agree with me that the publication uses garish and overdesigned pages, with a look and feel that bears little resemblance to a real magazine. Guess that just wasn't what you were looking for.

Designingly,
Corbett

William Neukom, vice president,
Legal Microsoft
Bill,
I'm glad that you were able to use the Microsoft Windows' screen dumps that I sent you for your Windows 2.03 legal defense. I see that you agree with me that the product encourages garish and overdesigned screens, with a look and feel that bears little resemblance to the Macintosh's. Guess that was just what you were looking for.

Legally,
Corbett

Fred Davis, editor-in-chief
MacUser
Fred,
Fine "HyperTalk" column, but the cowboy hat is strange. And where's Doug Clapp?

Applaudingly,
Corbett

Michael Swaine, editor-at-large
Dr. Dobb's Journal
Mike,
You've been looking a little tired lately. I think you're stretching yourself too thin. Too many writing assignments. No good will come of it, Mike, mark my words. One of these days you're going to be tapped out when the copy comes due, and then what'll you do? By the way, there are several other letters on this disk. Could you see that they get to the people to whom they're addressed?

Relatively,
Cousin Corbett

Michael Swaine

Michael Swaine
editor-at-large

#142 AUGUST

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER

OPTIMIZING C COMPILERS

Comparing the fastest of the fast: including Watcom's new C 6.5.

MANAGING MULTIPLE MODULES IN DOS

A C DIAGRAMMING UTILITY

SECRET-KEY CRYPTOGRAPHY

ANNUAL C ISSUE

0 38351 16562 8 08

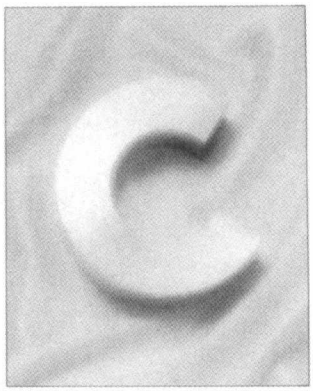
ARTICLES

C Compilers ▶ **Speed Trials: Five Cs Compared** **18**
by Richard Relph
Once again less is more as Rich compares this year's crop of hot C compilers by the likes of Borland, Computer Innovations, Datalight, Microsoft, and Watcom.

C Functions ▶ **Find That Function!** **30**
by Marvin Hymowech
Marvin explains two utilities he's written in Microsoft C 5.0 to build a function finder for MS-DOS that takes some of the grunt work out of revising your C source code.

Print Utility ▶ **Automatic Module Control in C** **40**
by Stewart Nutter
If managing multiple modules makes you mad, try Stewart's C print program. This potent utility not only prints a tree diagram but also checks for unused functions and recursion and determines the module size and line count.

A Tool for Secret Key Cryptography **50**
by John Michener
John uses a generalized rotor operator technique to construct a cryptographic secret key system that is strongly resistant to analysis.



REVIEWS

EXAMINING ROOM **56**
coordinated by Ron Copeland
In this month's offering Ernie Tello explores how well Objective-C's object-oriented approach improves handling conventional programming chores.

COLUMNS

C PROGRAMMING **98**
by Al Stevens
In his first appearance as a *DDJ* columnist, Al starts his list of C programming "crotchets" and discusses confusing, but necessary, C constructs.

RISC ▶ **THE FORTH COLUMN** **114**
by Martin Tracy
Martin is our real-time connection this month as he focuses on Forth-based RISC chips and an upcoming real-time programming contest that you may want to enter.

Pascal ▶ **STRUCTURED PROGRAMMING** **120**
by Kent Porter
Kent replaces DOS's inadequate *DIR* with a powerful new *SUB* command and offers some techniques for manipulating directories in Turbo Pascal 4.0.

PROGRAMMING PARADIGMS **130**
by Michael Swaine
Once again Michael attempts to go in many directions at once with a discussion of parallel algorithms—coupled with practical examples.

FORUM

EDITORIAL **6**
by Jonathan Erickson
RUNNING LIGHT **8**
by Tyler Sperry
LETTERS **12**
by you
CARTOON **12**
by Joe Sikoryak
SWAINE'S FLAMES **160**

PROGRAMMER'S SERVICES

ADVERTISER INDEX **145**
where to go for more information on products.
OF INTEREST **152**
brief product descriptions
PROGRAMMER'S MARKETPLACE **157**
classified ads

Next Issue

In September we'll take a look at software engineering, focusing on language-centered environments. We'll also examine how Ada measures up against C and Pascal in specific programming situations and how some of the newer Modula-2 compilers measure up against each other.

EDITORIAL

A few months back, Larry Rosler raised some eyebrows and turned some heads when he told a roomful of predominantly C programmers that if they wanted to see what C might look like in the future, they should look at Ada today. The point Rosler was trying to make is that he thinks C as we know it has been pushed to the limit of its capabilities.

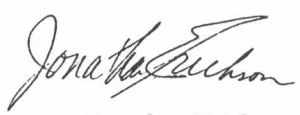
In explaining what he meant, Rosler, who is manager of Hewlett-Packard's Computer Languages Laboratory, began by describing the life cycle of C as progressing from a period of childhood and adolescence in the 1970s (characterized by Unix, portable C compilers, standard I/O libraries, and the publication of Kernighan and Ritchie's "white book") to a period of adulthood and maturity in the 1980s (characterized by PC-based compilers, dozens of implementations, and the development of C++). The late 1980s are particularly significant, Rosler said, because of the emergence of what he called "heavy" implementations of C. From the late 1980s on, however, Rosler was less optimistic about the future of the language. "C is old," he said, and in the early 1990s, we will see it enter a period of "senescence."

Among the reasons Rosler cited as contributing to the metamorphosis of C are the language's inability to cope effectively with deficiencies such as the scoping of symbols in large programs; the inability to handle new data types (long integers, complex numbers, and so forth); the lack of linguistic support for concurrency; and the deficiencies in I/O libraries. Rosler acknowledged that work is being done in many of these areas to compensate for current inadequacies with most of the work revolving around an ANSI Standard C and C++.

"These proposals," Rosler said, "will make an effective jump in the usefulness of C." However, he bluntly told the group that included Bjarne Stroustrup, "C++ isn't enough."

Whether or not you agree with Rosler about what future incarnations of C might look like, there's little question that the language will evolve and that compiler developers will continue to look for ways to heighten performance. This movement is reflected by the current push toward "optimizing" compilers. According to feedback we've been getting from readers, nearly 75 percent of those of you who have responded to our Reader Service card here in the magazine indicate that C is the language you use the most. This means that *DDJ* will continue to provide thorough coverage of C and share useful C tools with you no matter what the language eventually looks like.

Speaking of topics that will be covered in the future, we're starting to look at subjects for the next year and need to know what you think we should be covering. Some of the themes and articles we're considering include algorithms, programming in parallel environments, real-time programming, and object-oriented programming, to mention a few. If you have any preferences for general topics or specific articles, let me know. Associate editor Ron Copeland and I will be phoning quite a few of you to pick your brains, and we've been posting some of the articles that have been proposed to us recently in the DDJ Forum on CompuServe. Take a look there and tell us which ones you'd like to see in print.



Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Editor *Tyler Sperry*
Managing Editor *Monica E. Berg*
Senior Technical Editor *Kent Porter*
Associate Editor *Ron Copeland*
Assistant Editor *Sara Noah Ruddy*
Technical Editor *Richard Relph*
Contributing Editors *Al Stevens, Martin Tracy*
Copy Editors *Lorraine Aochi, Kevin Shafer, Rhoda Simmons*
Editor-at-Large *Michael Swaine*
Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Lynn Sanford*
Typographers *Lorraine Buckland, Margaret Anderson*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Asst. Circulation Manager *Andrea Weingart*
Newsstand Manager *Sarah Frisbie*
Direct Marketing *Kathleen Shay*
Fulfillment Coordinator *Francesca Martin*
Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*
Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, with two special issues per year by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Associate Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Request: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3713, Escondido, CA 92025. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA 800-321-3333; in CA 619-485-9623 or 566-6947. For book/software orders call 415-366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $28 per year airmail or $11 per year surface. All other countries add $32 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

August, as you're no doubt aware, is the month for *DDJ*'s annual C programming issue. The C environment has changed dramatically in the last few years. It wasn't too long ago that *DDJ* was printing the source code for Ron Cain's Small C compiler, not only because it was interesting to tear into a compiler and see how it worked, but also because many of our readers couldn't afford to buy the C compilers available then.

Times have changed, and we change with them. Although it is even easier today to write your own C compiler, most of the incentive has gone. Inexpensive compilers like Turbo C have made it possible for many of us to move on to the next stage: developing tools and programs for the next phase of the revolution. This issue reflects some of the advances in the C world.

Last year's special C issue had articles on optimization technologies in C compilers and a report on the approaching ANSI standard. At press time this year, there's still no final word on the standard. Optimization in C compilers, on the other hand, seems to have taken off. The last year has seen a number of company executives making contradictory claims about which compiler really generates the fastest and tightest code. In our lead article this month, *DDJ* technical editor Richard Relph examines five pretenders to the throne and reports on which *really* has the fastest code. We also have a couple of utilities for C programmers, to make the human part of the equation a little faster as well.

The times change, and we change with them. This issue marks the arrival of a new C columnist, Al Stevens. Al is the author of several programming books, including *C Data Base Development* and *C Development Tools*, and he has some interesting columns planned for the next few months.

This issue also marks the change in title for technical editor Kent Porter. By hook, crook, and vast karmic debt, we have managed to lure Kent into joining the *DDJ* staff full-time as our senior technical editor. Kent will still be writing "Structured Programming" and an occasional article, but he'll also be in the office helping us keep the technical level of the articles as high as possible.

The reading selection of the month is a bit different this time out. This month I'd like to suggest you check out Barbara Garson's new book, *The Electronic Sweatshop* (Simon & Schuster). The subtitle of the book, "How computers are transforming the office of the future into the factory of the past," might mislead you into dismissing this book as something for sociologists or computer science academics, but that'd be ignoring the book's real worth. The issues raised in this book are worthy of consideration by anyone who sees personal computers as instruments for liberating the human mind and spirit. Highfalutin words perhaps, but I'd suggest that it was our interest in liberating the mind that got most of us into computers in the first place.

The world changes, and we're part of the change. This issue marks my final appearance as *DDJ*'s editor. With Kent's arrival I feel confident of a continued high technical level for *DDJ*, even at the same time I'm excited about my return to freelancing. While my work at *DDJ* has been rewarding, it has also been consuming, and I look forward to completing some long-delayed programming projects. As my friend Bruce Webster likes to say, see you on the bitstream. . . .

Tyler Sperry
editor

ARCHIVES

Ten Years ago in *DDJ*

"The conservation view of a microcomputer system is that it should differ from large systems primarily in scale, in the way that a VW differs from a Mercedes, and that there ought not to be qualitative differences in hardware, software, or methods of operations. Innovations should arise in the giant systems, and then (to the extent that the micropocketbooks of micro users allow) be copied at the lower levels. I prefer another analogy. There are no elephants that burrow underground like moles or can fly like bats, and downsizing to the level of insects allows even greater diversification. Micro systems are going to be individualized and customized in ways that would horrify IBM, and some innovations at the micro level (most will of course be failures) may well migrate upward."—*"Letters,"* DDJ, *September 1978.*

What Will We C?

"C has continued to develop in recent years, occasionally by restrictions against manifestly nonportable or illegal programs that happened to be compiled into something useful. . . . It is more difficult, of course, to speculate about the future. C is now encountering more and more foreign environments, and this is producing many demands for C to adapt itself. . . . What is not likely is a fundamental change in the level of the language. Realistically, the very acceptance of C has compelled changes to be made only most cautiously and compatibly. Should the pressure for the improvements become too strong for the language to accommodate, C would probably have to be left as is, and a totally new language developed."—*D.M. Ritchie, S.C. Johnson, M.E. Les, and B.W. Kernighan, "The C Programming Language,"* DDJ, *May 1980.*

Now if it Could only be Memory Resident . . .

"Programmers often need to manually verify the results obtained by a computer, especially while sitting at a console and debugging code. Pocket calculators with capabilities like hex or octal readouts and Boolean functions are a convenient tool, but ignore the fact that the computer itself can operate as a handy desk calculator."
—*Mike Gabrielson, "Whatis: An Expression Evaluator for the 8080,"* DDJ, *October 1978.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Aid for Apple CP/M System

Dear *DDJ*,

Please help me with the necessary information to write a device driver for my Apple CP/M system. I have the Softcard and the ALDS system. Since I am using Elite Two double-sided 5.25-inch drives made by now defunct Rana Systems, the system has been modified to recognize these drives. The Ultraterm card by Videx gives the system an 80-column display. The Prometheus Versacard gives the system other capabilities.

I have recently acquired a Universal Disk Controller Card and 3.5-inch drive, both made by Central Point Software. I need to write a device driver or get one from Microsoft to use the 3.5-inch drive from my Apple CP/M V2.2 56K TPA system running on an Apple II+.

With six years experience in Z80 assembly language and some in 6502 assembly language, your help should be all that I need to get the 3.5-inch drive on-line with Apple CP/M.

Gayle Lee Fairless
Huntsville, Ala.
CompuServe 71571,321

Experience Necessary

Dear *DDJ*,

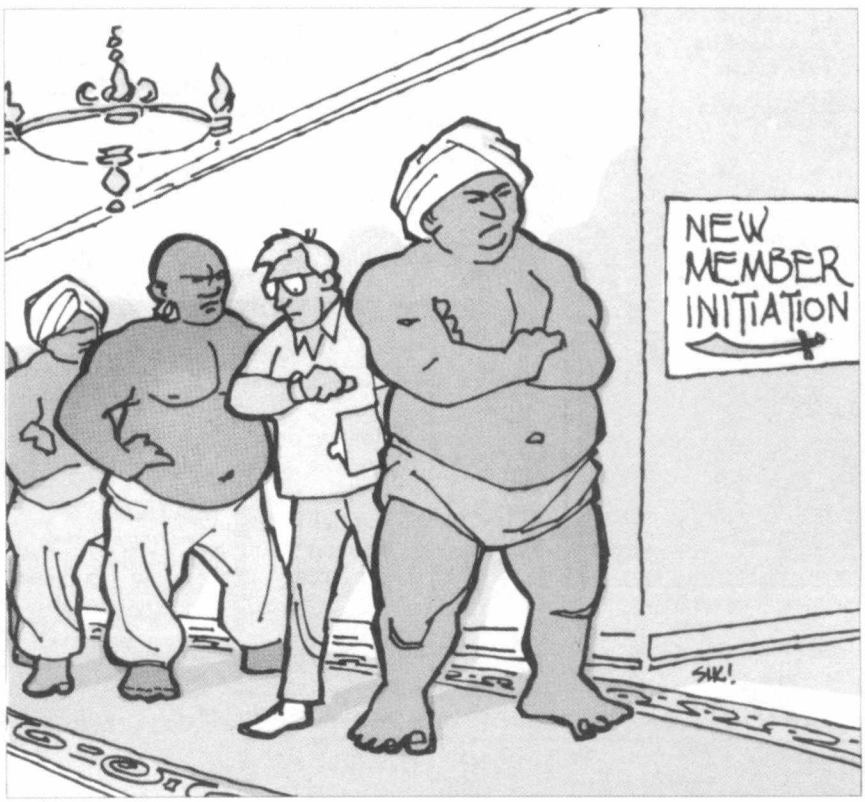
I was a little upset when I read the editorial in the June issue about the lack of good programmers. I believe the key word in the editorial is "experienced" programmers, not "good" programmers. Many of the companies that I have talked to will not even look at a programmer unless he or she has at least two years of experience.

I graduated from a small college with a BS in computer science in 1987 and have spent over a year so far looking for a position in the programming field. I have had about 10 to 15 interviews but have always found out that another person was hired because he or she had more experience. I always figured that if I became fluent in a few languages (C, Cobol, Basic, Pascal), that I could easily move from one system to another. After all, there are many similarities between computers and their operating systems; it's all a matter of semantics.

My response to those companies who are looking for people with two years of experience is: look harder at the entry level candidates. College work is experience! A college teacher told me that there was a certificate available from some computing society (ACM?) after four years of experience. He also stated that a degree in the computer science field could be substituted for two years of that experience. If this is true, why are recently graduated college students considered as having no experience? I am sure that they all worked hard to earn their degrees.

Entry level people trying to get into the business may cost the company less in salaries and will probably benefit the company if they stay for a few years. Not only do they gain experience but they will know the company and its systems better. It would also help the present staff: what better way is there to test your knowledge and your experience than to answer the ad hoc questions that an eager trainee can come up with?

Start a training or intern program that brings in entry level people. An intern program would be a great way to test the abilities of the candidates for the training program. This would benefit the students by giving them a few months of hands on experience that everyone is looking for. Many entry level people may not need all the training you think they will. Some will catch on to the company's policies and the computer systems in a very short time. Those are the people



Eventually, Ernie would realize that this was not a UNIX meeting after all.

who will benefit the company the most. A rigorous training class may not be needed. Simply assigning an employee to a trainee as a mentor would be more than adequate for most. The reduced salary that these trainees collect over an experienced candidate will save the company some money.

I am still looking for a position in the computer field. I did not spend four years of my life and over ten thousand dollars to give up and go to work the rest of my life as a laborer. I consider myself a good programmer and try to keep up with what the industry wants. Your magazine has been a great help. Get the companies to hire entry level people right out of school.

Gary Krone
Kansas City, Mo.

Machine Code Functions

Dear *DDJ*,

My company manufactures a PC compatible data-base software product. We are familiar with the PC environment at the user level. The vast majority of the work we do is microprocessor development in embedded control applications. The hardware we build is not PC compatible, however, so what I am anxious to find are resources for algorithms in general terms that can solve problems we often encounter.

There are many functions that are taken for granted in a high-level language that do not exist in machine code. For instance, floating-point routines, sine, cosine, square root, graphics, and ASCII to binary routines are becoming much more important to our needs. Where can the fundamental algorithms be found to implement these functions?

The ideal solution for me would be a reference book (or books) that describes at a flow chart level how to code these algorithms independent of a specific processor. The chart below shows some of the algorithms I would like to have in my bag of tricks.

If your readers could suggest places to look or publications that answer some of these questions, I would appreciate it very much. I think this need is shared by a great many engineers in addition to myself.

Alan Clark
Dynatech Nevada
Carson City, Nev.

floating point	add, subtract, multiply, divide
trig	sine, cosine, tangent
square root	integer and floating
integer to floating	conversion
floating to integer	conversion
graphics	how to model a bit mapped plane and draw lines from point to point, circles, and other shapes
ascii to integer	convert an ascii string to signed/unsigned integer
ascii to floating pt.	convert ascii string to signed/unsigned floating pt.

The Beat Goes On . . .

Dear *DDJ*,

The Turbo C versus Quick C debate eventually comes down to whether you like the Turbo or Quick interface better. I think that given time, the compilers will be about equal. At the moment, however, TC seems to have the edge as being a more complete, general purpose package than QC. TC provides about the same capability and performance as MSC 5.0 and

(continued on page 150)

SPEED TRIALS:

FIVE Cs COMPARED

by Richard Relph

C is a language renowned for its efficiency. This fact has not been lost on compiler vendors, who in their advertising claims are always using words such as *faster* and *fastest*. As the author of "Optimizing Compilers for C"[1] in last year's C issue of *DDJ*, I am well aware of the difference between advertising claims and the quality of generated code. This year, instead of looking at the theoretical aspects, I decided to take a look at the optimizing C compilers that are actually available.

This article reviews five optimizing C compilers for MS-DOS—Microsoft C, Version 5.1; Watcom C, Version 6.5; Borland Turbo C, Version 1.5; Datalight's Optimum-C, Version 3.12; and Computer Innovations C86+, Version 1.10. They are all from vendors who make vast claims as to the efficiency of their compilers. I decided not to include MetaWare's High C compiler, not because of a lack of technical merit but because my company now has a close business relationship with MetaWare and I didn't want any charges of bias.

With a previous review[2] for *DDJ*, there'd been a few accusations from readers that I was supplying an overwhelming number of numbers. This review will not suffer that fate. Instead, I'll concentrate on usability issues, language conformance, and the quality of the generated code.

Richard Relph is a technical editor for DDJ *and the author of several previous articles on C compilers. Richard is a member of the ANSI X3J11 committee, and the chief technical officer of Embedded Performance Inc. He may be reached at EPI, 3707 Williams Rd., Ste. 2, San Jose, CA 95117.*

(continued from page 19)

The Test Software

This year, my company splurged and purchased the C Torture Test from Austin Code Works for $20. (See Sidebar.) Unfortunately, it seemed evident from the test results that most of the C compiler vendors never bothered to invest the $20 to obtain this suite.

I adapted the test suite for ANSI compatibility. The final performance tests consisted of compiling, linking, and running both the modified C Torture Test and the Dhrystone benchmark.[2] (As most *DDJ* readers are aware, the Dhrystone is a performance benchmark that times

Optimizing Techniques

The term "optimization" refers to the techniques a compiler uses to produce code that is appreciably smaller and/or faster than code generated by comparable non-optimized compilers. Over the past year, C compiler manufacturers have been jumping on the optimization bandwagon as a means of both improving their product and of differentiating it from other compilers on dealers' shelves. Unfortunately, they don't go into much detail explaining exactly what they mean by optimization other than saying their product is "faster and better" than other available compilers.

There are several optimization techniques available to compiler developers of which one or more method may be implemented within an individual compiler. These techniques include register allocation, constant propagation, dead assignment elimination, dead code removal, copy propagation, and common subexpression elimination. For a detailed description of each of these methods, see "Optimizing Compilers for C" by Richard Relph (*DDJ*, August, 1987). You may also want to refer to *Compiler Principles, Techniques, and Tools* (also known as "The Dragon Book") by A. Aho, J. Ullman, and R. Sethi (Addison-Wesley).

Of all these methods, register allocation is perhaps the most advanced and important optimization technique and a key aspect of approach used by developers such as Watcom, Datalight/Zortech, and others. With register allocation, the number of memory references in a program are kept to a minimum by keeping variables and temporary results in registers. This allows shorter instructions to be generated which results in smaller, faster code.

The compilation process always begins by translating source into some internal, intermediate level representation. Once an intermediate representation of a function has been built, algorithms can be used to perform live variable analysis. A variable is live at a point in the flow graph if its value at that point could be used at some other point in the flow graph. Intuitively, a variable is live if its value may be used "later on" in the function.

Live variable information is used to build a data structure called the conflict graph. Every variable in the function has a corresponding node in the conflict graph. A link between two nodes is present whenever two variables are live at the same point in the flow graph. These links indicate that the two variables may not share the same register. The method used to assign registers to variables is called a coloring algorithm since it is similar to the problem of coloring a map so that no two adjacent countries have the same color. Since a machine has a limited number of registers, the object is to color (assign a register to) as many nodes of the conflict graph as possible without assigning the same color (register) to any two adjacent nodes. When a node is left uncolored, the corresponding variable must be assigned a position in memory or on the stack.

Of course, allocating registers intelligently requires information about the use of the variables. A variable that is heavily used in a loop is a better candidate for a register than a variable that is only used twice at the beginning of a function. Analysis may be performed on the flow graph to determine the loop nesting depth of each basic block. The uses of each variable are found and a weight is assigned to each use. Higher weights are assigned to uses within one or more loops. These weights are added up and stored in the conflict graph. The coloring algorithm takes these weights into account when assigning registers.

Although optimization can occur over several ranges—statement, block, function, module, and program—in the compilation process, the most advanced are global optimization techniques that occur over the range of an entire function rather than just a single statement. Global optimizations are applied to intermediate *representations* of functions that consist of instructions formed by a set of operands and one operator. These instruction sets are organized into sequences called blocks that have one label at the beginning and one jump or return at the end. Blocks are linked together to form the control flow of the function.

In some cases, it is convenient for compilers to let the register allocator introduce some anomalies that are removed later in a post-optimization phase. One post-optimization technique involves keeping track of the value of a register at any point in the block. The data structure used to keep track of this information is sometimes called a register scoreboard. Each instruction is examined and the scoreboard is updated to reflect the effect that the instruction has on the register. If a register is stored in a memory location, the scoreboard remembers that the memory location and the register contain the same value. When a move instruction is encountered, the scoreboard is consulted to determine if the move is redundant. When a memory reference is found in an instruction, the scoreboard is checked to find out if a register contains the appropriate value. By carefully designing the register scoreboard, a significant gain can be achieved. — eds.

(continued from page 20)

execution of an "average" set of instructions and is as much a benchmark of characters and interger performance as Whetstone is of floating-point performance. Around the time of this review, I was also attempting to port a very large source-level debugger (Third Eye's CDB) from Unix to the PC. As a result of these different exercises, I soon had an excellent feel for all the compilers.

I ran all the tests on a Compaq 386 (16-MHz) Model 130 with 3 Mbytes of memory. I used 1 Mbyte of memory as a RAM disk (using RAMDRIVE which comes with Microsoft Windows/386) and the other spare Mbyte as a disk cache (using SMARTDRV which also comes with Microsoft Windows/386). I believe that machines with this speed and memory are no longer exceptional and will soon be the norm for most developers. With one exception (mentioned later), the test results should be representative of the rankings for these compilers, regardless of the particular machine you use. If you're impatient for the results, you can grab a peek at Table 1, below.

Although I used a 386 machine for the tests, please note that I made no effort to have the compilers produce code specifically for the 386. The code was generated to run on any member of the Intel 8086/80286/80386 line. As the Dhrystone was used as the performance test, all the compilers were directed to generate code with the fastest execution time (instead of optimizing for space).

Common Problems

In the C Torture Test, there were a couple of constructs that gave more than one compiler problems. The first is the following sequence of code:

```
foo(a)
struct x { int y, z; };
struct x a;
{ . . . }
```

This source code should declare a structure tag *x* and define a function *foo* taking an argument *a* with type *struct x*. Three of the five compilers reviewed here, however, would not compile this construct.

The second common problem has to do with large constants. The suite attempts to confirm that decimal, octal, and hex representations of the same value all yield, in fact, the same value. The suite tests this at many points in the number range—nearly all of which work fine, of course. The problems arose when the suite attempted to test compile-time constant overflow. In particular, the test suite attempts to create three numbers greater than or equal to 2^{32} in each of the three bases. Several of the compilers refused to compile such constants at all, yielding fatal errors. I believe the appropriate response is to warn about such things but also to continue compilation.

The last area of common problems has to do with the new ANSI escape sequence \x, which is a lead-in to a hex constant. The problem is what should the compiler do if all that it sees is \x? Should it yield a 0, a lowercase x, or a compile-time error? The ANSI draft is mute. My preference, as both a programmer and as a member of the ANSI committee, is to have \x yield a lower-case x, as this is unambiguous and will break the least amount of existing code.

The highlights (if you can call them that) of the test suite output are shown as Examples 1–4, starting on page 24. Perhaps the most interesting thing to note from the test results is how the compilers differ in allocating registers. Watcom C's output, for example, shows that this compiler reserves four(!) registers for pointer variables.

Microsoft C

The current standard bearer isn't just resting on its laurels. Microsoft C 5.1 improves on the old Version 4.0 C by including the Quick C compiler, an improved CodeView, the ability to run under and/or generate code for OS/2, as well as better code generation. The package now contains 14 diskettes, 2 of which are high density (the OS/2 library disk and the OS/2 CodeView disk). Regrettably, the documentation in the package I got was the 5.0 documentation with 5.1 update pages. The new update pages do not fit in the 5.0 binders, leading to some real headaches.

Microsoft C was able to deal with structure tag declarations in the area between a function declarator and its opening bracket. On the \x issue, Microsoft's compiler chose to generate a fatal compile-time error. When it came to constants that were larger than *unsigned long*, the compiler gave error messages. This is acceptable. What is not is that the compiler also complained about the decimal form of the largest *unsigned long*, claiming

Product	Version	Compile/ Link/Run (sec)	Dhrystone (sec)
Microsoft C	5.1	129	10.4*
Watcom C	6.0	187	10.6
Borland Turbo C	1.5	43	13.3
Datalight Optimum C	3.1	165	10.9
Computer Innovations C86+	1.1	327	14.0

*Time was reduced to 10.0 seconds with changed optimization arguments. See text.

Table 1: *Performance test results*

(continued from page 22)

the constant was too big. The ANSI draft clearly states that this constant should compile, with a resulting type of *unsigned long*.

I originally compiled the suite and benchmarks with */Ox* (which is equivalent to */Oatil /Gs*), which yielded a number of validation suite errors when aliasing was being tested explicitly. So I decided to run the compiler for this review with */Oilt /Gs*, which eliminates */Oa* from the */Ox* option. I was surprised to find that, besides all the aliasing tests now working, some of the floating-point tests that had previously failed (with */Ox*) now succeeded.

When compiling with */Ox*, Microsoft seems to cheat on floating point a bit. Under these conditions, it seems that 1.084202e–019 is the smallest number that can be added to either a single- or double-precision 1.0 and result in a number that does not compare equal to 1.0. Because *float* variables have only 24 logical mantissa bits, it's just not possible for there to be 19 digits of significance. This result could be obtained only if the compiler skipped the assignment to the *float* or *double* variable in the loop instead of leaving the result in the 80-bit floating-point registers. This same sort of "cheating" also explains the *s626,er1* error I had got with */Ox* that went away with */Oilt /Gs*.

Microsoft C also doesn't seem to support signed bit fields yet, as required by the ANSI standard.

As for performance, it took 2 minutes and 9 seconds to compile all the test suite, run it, and then compile the Dhrystone and run it too. The execution time for the

```
  8 bits in chars.
 16 bits in ints.
 16 bits in shorts.
 32 bits in longs.
 16 bits in unsigneds.
 32 bits in floats.
 64 bits in doubles.

1.192093e-007 is the least number that can be added to 1.
                                                    (float).
2.220446e-016 is the least number that can be added to 1.
                                                   (double).

Register count for char is unreliable.
Register count for pointer is unreliable.
Register count for int is unreliable.

char alignment: 1
short alignment: 2
int alignment: 2
long alignment: 2
unsigned alignment: 2
float alignment: 2
double alignment: 2

No errors detected.
```

Example 1: *Test suite results for Microsoft C, Version 5.1*

```
  8 bits in chars.
 16 bits in ints.
 16 bits in shorts.
 32 bits in longs.
 16 bits in unsigneds.
 32 bits in floats.
 64 bits in doubles.

1.192093e-007 is the least number that can be added to 1.
                                                    (float).
2.220446e-016 is the least number that can be added to 1.
                                                   (double).

sign extension in char assignments

0 registers assigned to char variables.
2 registers assigned to pointer variables.
2 registers assigned to int variables.

char alignment: 1
short alignment: 2
int alignment: 2
long alignment: 2
unsigned alignment: 2
float alignment: 2
double alignment: 2

Sign extension in fields
Be especially careful with 1-bit fields!

No errors detected.
```

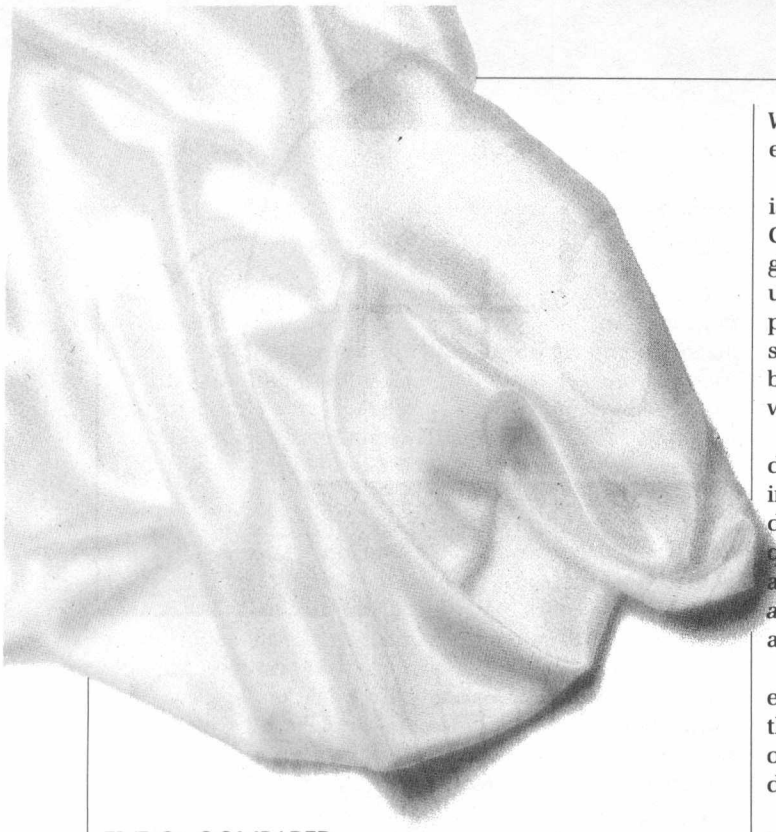
Example 2: *Test suite results for Watcom C, Version 6.0*

```
  8 bits in chars.
 16 bits in ints.
 16 bits in shorts.
 32 bits in longs.
 16 bits in unsigneds.
 32 bits in floats.
 64 bits in doubles.

1.192093e-007 is the least number that can be added to 1.
                                                    (float).
2.220446e-016 is the least number that can be added to 1.
                                                   (double).

Register count for char is unreliable.
4 registers assigned to pointer variables.
Register count for int is unreliable.

char alignment: 1
short alignment: 1
int alignment: 1
long alignment: 1
unsigned alignment: 1
float alignment: 1
double alignment: 1

Sign extension in fields
Be especially careful with 1-bit fields!

Failed.
```

Example 3: *Test suite results for Borland's Turbo C, Version 1.5*

```
  8 bits in chars.
 16 bits in ints.
 16 bits in shorts.
 32 bits in longs.
 16 bits in unsigneds.
 32 bits in floats.
 64 bits in doubles.

1.192093e-007 is the least number that can be added to 1.
                                                    (float).
2.220446e-016 is the least number that can be added to 1.
                                                   (double).

1 register assigned to char variables.
Register count for pointer is unreliable.
Register count for int is unreliable.

char alignment: 1
short alignment: 2
int alignment: 2
long alignment: 2
unsigned alignment: 2
float alignment: 2
double alignment: 2

Failed.
```

Example 4: *Test suite results for Datalight's Optimum-C, Version 3.12*



(continued from page 24)

Dhrystone was the second best of all the compilers at 10.4 seconds. When compiled with */Ox*, the Dhrystone time was 10.0 seconds, but I'm not certain the program ran correctly, given the problems with */Oa* noted earlier.

Watcom C

Watcom is a newcomer to the DOS C world, but its first effort in the arena is very impressive. Watcom C gives no quarter to Microsoft C. The code generation is exceptional, and the package is extensive. Included with the optimizing compiler is a Quick C-like compiler (called Express C, also available separately), a source-level debugger that is better in some ways than CodeView, an editor, and the usual complement of tools. Watcom C comes on eight diskettes with five manuals—four wire-bound and one perfect-bound (like *DDJ*).

Watcom C failed to accept a structure declaration between a function declarator and its open {. It also complained about \x. The character constant \X is also subjected to the same treatment as \x, which is inexplicable as C has never had case-insensitive escape sequences. This is an outright bug. Watcom C did successfully deal with constants that overflowed 32 bits but did so without warning (at any warning level).

Watcom enforces a concept endorsed by the ANSI committee that may cause some of your existing code to break. The concept is called Miranda prototyping, after the Miranda rule of Supreme Court note. The idea is that if you do not supply a prototype for a function, one will be supplied for you. In particular, if a function *foo* is called in the absence of a prototype, it is assumed that all calls to *foo* will pass arguments of the same type, number, and order as the first. If you do not obey this rule, you fall into the ANSI pit of undefined behavior. With Watcom C, such breaking of the rules is a fatal run-time error. The reason why it's a fatal error is very interesting.

Watcom C uses, by default, a calling convention that is different from that used by either Microsoft C or Lattice C. Watcom C passes parameters in registers, which is a good idea but which requires the consistency of function usage mandated by ANSI for all "strictly conforming" programs. It is important to note that Watcom C can switch calling conventions on a function-by-function basis, giving you the maxi-mum flexibility imaginable in working with other compilers (regardless of language).

As a result of Watcom's use of registers, its compiler discovered inconsistencies in arguments to a function in the test suite. I prefer this behavior because it will detect code that certainly isn't ANSI conforming and quite probably is wrong. In the specific example, the arguments to the function were *pointer to int*, *pointer to array of int*, and *pointer to array of array of int*—all amounting to *pointer to int*.

After the compilation errors were cleaned up, the execution showed a small flaw in floating point (*s626,er1*) that is more probably an invalid assumption on the part of the test than a compiler bug. Watcom does fold duplicate string constants into a single string.

Watcom C took 3 minutes and 20 seconds to compile, link, and run the validation suite and Dhrystone. Dhrystone run time was was by far the best of the bunch at 8.8 seconds, or 5681 Dhrystones per second. The difference in compile time is probably mostly because of the lack of use of the available RAM disk for temporary files.

Borland's Turbo C

Borland's Turbo C excels in two areas: speed of compilation and Unix compatibility. Code generation is not so great, and the lack of a source-level debugger is fatal.

Turbo C failed to accept a structure declaration between a function declarator and its opening curly bracket but properly handled excessively large constants and \x. After the problem with structure declarations was fixed, no problems were reported at all—a very impressive feat.

Not so impressive, however, is the lack of a clock function in the library. Borland's was the only compiler that did not have this key (for benchmarks) function.

Borland's compiler really shines in compile and link times. Turbo C took only 43 seconds to compile and link—something that took more than 2 minutes for the nearest competitor. The Dhrystone run time was a respectable, if not overwhelming, 13.3 seconds.

Although it has long been rumored that the next version of Turbo C will include debugging support, Version 2.0 wasn't available from Borland (not even in beta) at the time I conducted my tests. Borland did give me a demonstration of some planned improvements to Turbo C at its offices but declined to let me examine the product in its unfinished state. All I can say at this point is that it appears that the next version of Turbo C will include some tweaks to the run-time library and, more important, the long-awaited source-level debugger. I did not see any evidence, however, of major improvements in code generation—and it will take some real work for Borland to seriously challenge Watcom and Microsoft in this area.

Datalight's Optimum-C

Datalight's Optimum-C features excellent code generation and is pretty quick in compilation. But the package suffers from the lack of a source-level debugger.

Optimum-C was able to deal with structure tag declarations in the area between a function declarator and its opening curly bracket, and with the \x issue but not with the very-large-constant test. It produced the wrong answer for the size of a string constant of odd length by rounding it up. This resulted in a character-by-character memory compare looking at one too many bytes, the last of which, of course, failed.

Datalight's compiler took 2 minutes and 45 seconds to compile, link, and run the validation suite and the Dhrystone. Dhrystone run time alone accounted for 10.9 seconds—the third best of the bunch.

My recommendation to Datalight would be somehow to include a debugger to round out an otherwise complete package.

Computer Innovations' C 86+

Computer Innovations was the first company to advertise an optimizing compiler as such. C86+ is the product. Unfortunately, the compiler is slow and the code it produces is not so great. Aggravating the problems of performance was the lack of a source-level debugger.

C86+ failed to accept a structure declaration between a function declarator and its opening curly bracket. After that was fixed, no problems were reported at compile time, but the validation suite ran two tests, then terminated with *READ*, whatever that means. Even odder was that when I ran the executable version of this test, I got the following output:

```
Section s22          returned 0.
Section s241         returned 0.
s243,er2
Section s243         returned 2.
READ
```

In contrast, when I redirected the output to a file, all I got was the last line. Neither result amounts to an acceptable performance on the test suite. C86+ took 5 minutes and 27 seconds to compile, link, and run the validation suite and the Dhrystone. Dhrystone run time accounts for 14.0 seconds. This makes C86+ the slowest compiler producing the slowest code in this group.

Summary

Because the emphasis here was on the quality of the code produced by the compilers, I didn't address other aspects of the packages. All the compilers reviewed here came with adequate documentation as well as an editor and a make facility. None of the editors even vaguely tempted me to switch away from Epsilon. Regarding the make utilities, Microsoft's was the weakest of the bunch but was fairly easy to understand; the other vendors attempted more or less to mimic the standard (Unix) make. Again, I found nothing to make me switch from the tool I already use (in this case, NDMake from D.G. Kneller).

As you read earlier, none of the compilers emerged from the testing unscathed. As a result, my recommendations really depend on the type of work you want to do.

If I were programming for OS/2 and DOS, then Microsoft is the only choice, primarily because of its superior debugger, CodeView. But if I were to program for DOS only, I would have great difficulty deciding between Microsoft and Watcom. At EPI we have been using

Putting C Compilers Through the Wringer

All the major players in the C compiler marketplace claim to support the standard language as defined by Kernighan and Ritchie, but do they really? And if not, what are the discrepancies?

An outfit called The Austin Code Works (11100 Leafwood Ln., Austin, TX 78750; 512-258-0785) sells a test suite to find out. It's called the C Compiler Torture Test. For $20, you get a PC-compatible disk containing 23 C programs and a READ.ME file that tells you how to use them.

The Torture Test measures a compiler's compliance with the K&R "white book.[3]" In a total of some 6600 lines of code, the suite exercises the entire standard. Particularly stressed are language features likely to be used by advanced C programmers: complex constructs, cascading defines, pointer manipulation such as multiple indirection, and so forth.

Testing with the suite occurs on two levels: compile-time and run-time. In some cases, the compiler might choke on certain constructs. This is a clear indication of noncompliance. A clean compile means that the compiler accepted the source language, but it doesn't mean that the result is correct.

Consequently, after torturing the compiler, you run each test program to see that it does what is expected. For example, dereferencing a chain of pointers should ultimately lead to a known variable that the program can compare with a hard-coded value. When the result is wrong, the program issues an error message. The messages are cryptic, but the test programs are structured so that you can look up the error message in the source text and determine from comments what the problem is.

Since the Torture Test programs contain only vanilla K&R code (regardless of its level of difficulty), a perfectly complying compiler should compile them without a whimper and the resulting programs should run correctly. A failure at either level of testing pinpoints lack of compliance with the standard.

The C Torture Test is a valuable resource to both compiler writers and C users. Although delivered on a PC diskette, the code can be ported to any platform and used as is. And at only $20, nobody can complain about the price. — eds.

Microsoft but we are going to start using Watcom (at least in parallel). It's a close call, though, as Microsoft still doesn't have a decent make and Quick C is crippled by the lack of support for multiple memory models, as is Express C.

Watcom C bears special watching as an excellent all-round compiler, good for DOS work, OK for porting work, and better than most for embedded systems work. I really want this compiler to succeed because it is so close to being the absolute best in every way. Watcom C is as good as Microsoft C in every significant way and is better in a few, most notably in execution time.

If I were porting existing programs from Unix to DOS, then Borland Turbo C would be my choice, but just barely. If the program is known to be portable, then Turbo C offers the best chance of running it unchanged. But the lack of a debugger makes using Turbo C problematical if the program doesn't "just run" after compiling. With the debugger I saw at Borland, this reservation would go away completely.

As nice as the Datalight Optimum-C compiler is, however, you probably won't be able to find it on the shelves much longer because the marketing rights to Optimum-C have recently been licensed to a British company called Zortech. According to a Zortech spokesperson, that company will not market Optimum-C per se, but instead incorporate the compiler's optimization technology into its own compiler called Zortech C. Datalight customers can expect to get product support from Zortech.

Datalight's Optimum-C is an excellent compiler, and I do like it very much. You get source code for the run-time library and an outstanding compiler for very little money. But the package lacks a debugger and professional documentation. Optimum-C would be my first choice for embedded systems work, but still it is not ideal (missing are the necessary utilities for burning PROMs and "remote" debugging).

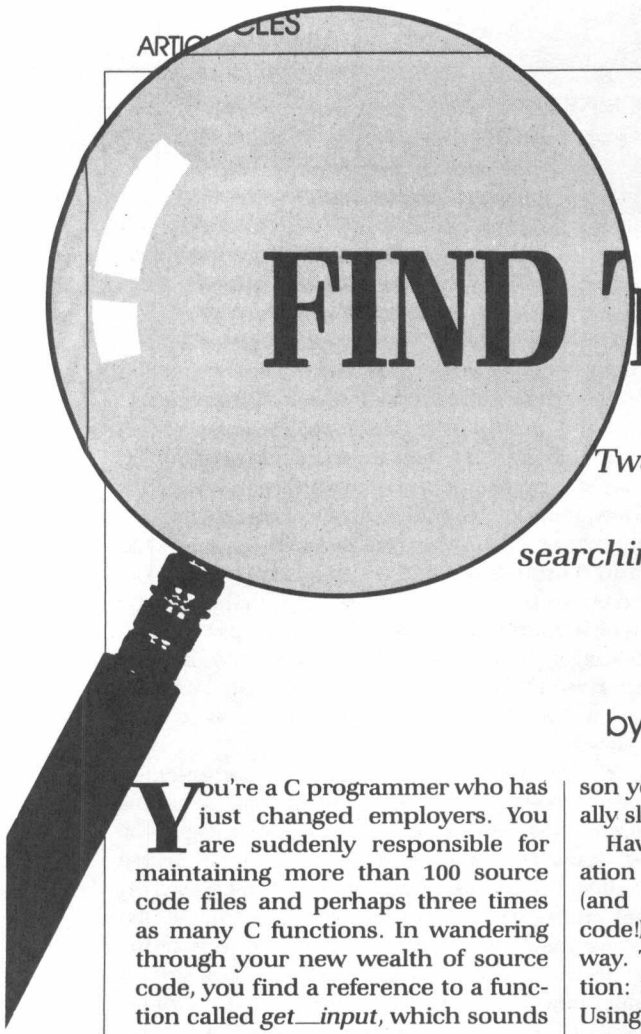
Computer Innovations' C86+, regrettably, has nothing except the reputation of the company to recommend it. You do get source code for the library but this hardly makes up for the lack of a debugger. And there's no escaping the fact that C86+ was the slowest compiler of the group, producing the slowest code.

Notes

1. Richard Relph, "Optimizing Compilers for C," *DDJ* (August 1987).
2. Richard Relph, et al., "Benchmarking C Compilers," *DDJ* (August 1986).
3. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs; N.J.: Prentice-Hall, 1978).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**



FIND THAT FUNCTION!

Two utilities that take the grunt work out of searching through C source code

by Marvin Hymowech

You're a C programmer who has just changed employers. You are suddenly responsible for maintaining more than 100 source code files and perhaps three times as many C functions. In wandering through your new wealth of source code, you find a reference to a function called *get_input*, which sounds like a good thing to look at next. But which file is it in? Or perhaps you recall seeing a function called *save_screen* (or was it *save_scrn*?) that would be just the thing to use in that new function you have to write, but you just can't recall where you saw it. What to do?

The traditional way of handling these problems is this: First, you use a text-searching program such as *grep* (Unix) or *ts* (from The Norton Utilities) and search through all your source files for every reference to the function you are looking for until you finally locate the reference that is the function definition. Then, you bring up the source file in your editor. Next, you search for the function again until you finally find the function definition. By this time, the reason you were looking for it has usually slipped your mind.

Having suffered through this situation several times over the years (and not just with other peoples' code!), I resolved to find a better way. This article presents my solution: a function finder for MS-DOS. Using the function finder (actually, it consists of two programs) speeds up the process considerably. You simply invoke the finder with the function name as an argument. The program then scans an index file, invokes your editor using the appropriate source file, and, for sophisticated editors such as Brief, even positions the cursor at the function definition.

The two programs that make up the function finder, *bldfuncs* and *getf*, are written in C. Although I used Microsoft C (Version 5.0) to compile the programs presented here, the code could be ported to other compilers (or to Unix) with minimal changes. The editor I use is Brief, Version 2.01, but any other editor could be used.

Marvin Hymowech works as a programmer for Condor Computer Corp. Previously he taught mathematics at the University of Michigan in Ann Arbor. He may be reached at 4906 Cole Blvd., Ypsilanti, MI 48197.

How It Works

As I mentioned earlier, my solution consists of two programs, *bldfuncs* and *getf*. *bldfuncs* constructs an index by reading through all the C source files specified on the command line and then constructing a text file (*funcs.txt*) that contains the name of each C source file read and a list of all functions defined therein. The resulting file looks something like this:

```
file_1.c:
    function_1a
    function_1b
    . . .
    function_1z;

file_2.c:
    function_2a
    . . .
```

Once the index file has been built by *bldfuncs*, *getf* can be run with a function name as a command-line argument. *getf* reads *funcs.txt* until it finds a match for the specified function name (wildcard characters are allowed) and then gets the name of the C source file in which the function resides. Next, *getf* constructs a DOS command line invoking the editor of your choice with this source file as the file to be edited. Then, a DOS *exec* call is performed to replace *getf* by the editor in memory. For editors such as Brief, which let you specify editor commands on the command line as well, even more is possible: You can specify an appropriate search command to position the cursor at an occurrence of the function name, which will hopefully be the actual definition of the sought-for function.

In brief then, you run *bldfuncs*

once to construct *funcs.txt* and thereafter whenever major changes occur in the distribution of functions among your source files. If you then need that elusive function *save_screen* (*save_scrn*?), you just type getf *save_scr**. One of three things will happen then:

1. *getf* will politely tell you that there is no such function in *funcs.txt*.
2. *getf* will find that there is exactly one function in *funcs.txt* matching the mask *save_scr**, and the next thing you will see is the desired source file in your editor, with the cursor positioned at the exact place where the function is defined (maybe).
3. *getf* will find more than one match for *save_scr** and will present you with a menu of such functions and the source files in which they are defined; after you choose one of these files, it will be presented in the editor, as in case 2.

bldfuncs is intended to read C files that work with almost any C compiler on the market; it is assumed, however, that the files will compile without errors.

Dealing with Editors

Now for some of the details. In an effort to land exactly at the function definition after the editor was invoked, I first tried instructing Brief to start at the beginning of the file and search forward for the function name. Most programmers, however, code so that functions are more likely to be referenced (in the source file in which they are defined) *before* they are defined rather than *after*. In keeping with this general rule, I wrote a custom Brief macro to start at the end of the file and search backward for the function name instead. This worked much better and in fact lands me at the exact spot about 50 percent of the time; when it doesn't work, I just use the key assigned to *search_again* to continue looking.

The command line *getf* uses to invoke the editor is built by using the DOS environment variable *GETFEDIT*, which allows users to customize *getf* to their editor of choice. On my machine, my autoexec.bat file contains the line:

```
set GETFEDIT=
        b -m'funcsrch %%s' %%s
```

Where *funcsrch* is the Brief macro mentioned earlier. The *-m* option tells Brief to invoke the macro in quotes after it begins. *getf* replaces the first *%%s* by the function name it is looking for and the second *%%s* by the file name in which it will be found.

My Brief macro *funcsrch* is shown in Example 1, this page. As you can see, Brief is programmed in a language that is roughly a cross between C and Lisp. The effect of this macro is first to position the cursor at the end of the file being edited and then to search backward for a specified string (the function name in this instance). I found this approach was more effective for positioning the cursor exactly on the definition of the sought-for function rather than on a reference to it. The built-in macro *end_of_buffer* positions the cursor at the end of the buffer and then the *search_back* macro is used to look for the string *s* obtained from the command line invoking the macro. The built-in external variable *_s_pat* is then set to the same value as *s* so that subsequent invocations of the macro *search_again* (assigned to Shift-F5 on my machine) will continue to search backward for the function name in question (also note that the external variable *_dir* is set to *0* so that subsequent *search_again*'s will go backward, not forward).

Building the Function List

bldfuncs (see Listing One, page 70) presented the thorniest problem: The program had to be sufficiently cognizant of C syntax to extract the names of functions defined in a source file yet ignore the similar constructions, such as prototype function declarations, used for argument type checking.

The approach I decided upon was to use several "filter functions" in succession to simplify the character stream obtained from the source file until I was able to extract the function names. Explaining from the bottom up (which is essentially the order in which the program was coded), the lowest-level filter is *filter_cmt*, which reads the raw character stream and returns a character stream identical to its input except that all comments have been elimi-

```
(macro funcsrch
   (
      ( string s )
      ( extern _s_pat _dir center_line )
      ( get_parm 0 s )
      ( message "locating function %s..." s )
      ( end_of_buffer )
      (if ( > (search_back s) 0 )
         (
            ( message "function %s found" s )
            ( center_line )
            ( sprintf _s_pat "%s" s )
            ( = _dir 0 )
         )
      ;else
         (
            ( top_of_buffer )
            ( beep )
            ( message "function %s not found" s )
         )
      )
   )
)
```

Example 1: *A search macro for the Brief editor*

```
bldfuncs.obj: bldfuncs.c
        cl /c bldfuncs.c

bldfuncs.exe: bldfuncs.obj
        link bldfuncs+\msc5\lib\setargv/ST:14000/NOE;
```

Example 2: *Make file for bldfuncs*

C FUNCTION
(continued from page 31)

nated. (Each filter function behaves like *fgetc* does—each takes a *FILE* pointer as input and returns characters, the value *EOF*, or other special values marking structures that have been "collapsed.") *filter_cmt* is essentially a small finite-state machine, the states reflecting whether you have just received an * (asterisk), or a / (slash), or neither. Some C compilers allow nested comments, so *filter_cmt* maintains a *cmt_level* variable, which is incremented when a /* (slash, asterisk) is received and decremented when an */ (asterisk, slash) is received; characters are returned only when *cmt_level* reaches zero.

The next filter is *filter_quotes*, which reads the character stream returned by *filter_cmt* and replaces any quoted string (delimited by either single or double quotes) by the special value *QUOTES*, which is used as a place marker for the original string. Higher-level filters look for matching curly braces, for example, and would be confused by curly braces occurring within quotes. The only subtlety in *filter_quotes* is to treat escaped characters (preceded by a backslash) correctly to avoid terminating a quoted string prematurely—for example, the string \} (slash, curly brace).

Next in the hierarchy comes *filter_ppdir*, whose task is to read from the stream provided by *filter_quotes* and eliminate all preprocessor directives. (I have made the simplifying assumption that no function will be defined either via a *#define* directive or in an include file. Although such peculiarities are possible, they are rare and are not constructs found when good programming style is employed.) The gotcha to be avoided in this filter is that *#define* constructs may extend to several programming lines, so *filter_ppdir* is careful to scan for escaped new-line sequences (a backslash followed immediately by a newline character) before deciding that a *#define* construct has terminated.

Next, *filter_curly_braces* reads the stream returned by *filter_ppdir* and replaces all characters between matching curly brace pairs ({ }) by the special value *BRACES*. This is accomplished by maintaining a brace count that begins at zero and is incremented when a left brace is encountered and decremented when a right brace is encountered. While the count is nonzero, no incoming character is returned to the caller.

Getting Down to Functions

At this point, the filtered input stream consists of external data items, function models for type checking, and actual function definitions. To simplify the task further, *filter_parens* reads the character stream provided by *filter_curly_braces* and eliminates all characters between parentheses, returning instead the special value *PARENS*. This approach eliminates thorny problems in the declaration of formal function parameters—for example, in the function definition:

```
int function1(a,b,c)
int (*a)( );
char (*b)( );
int c;
{
}
```

Some data initialization expressions can have constructions that resemble functions also—for example:

```
int size = sizeof(array)/
```

C FUNCTION

```
sizeof(element);
```

To avoid such problems, *filter_data* reads the stream provided by *filter_parens* and deletes the right-hand side of any assignment and the equal sign, leaving just the semicolon.

The stream returned by *filter_data* is sufficiently simple that it can now be used to extract the names of defined functions. The routine *get_names_one_file* opens the file specified by the parameter *source_file_name*; reads this file via *filter_data*; and writes the resulting function names to the file specified by the parameter *fp_out*, which is the already opened *FILE* pointer to the output stream for *funcs.txt*. This is done by storing characters from *filter_data* until *EOF*, or a semicolon, or the *PARENS* symbol is encountered. You are really only interested in sequences like this that end with *PARENS* because a sequence ending with a semicolon before a *PARENS* symbol cannot represent a function definition.

The routine *get_fn_name* is now used to extract the function name from the stored line, by scanning backward from the *PARENS* symbol until a character is encountered that is neither an underscore nor an alphanumeric. The decision as to whether this is a function definition or a type-checking construction is made as follows: If the first non-white-space character encountered after the *PARENS* symbol is a semicolon or a comma, it is a type-checking construction. The comma might occur as follows:

```
int func1(int, char), func2(int);
```

If you have an actual function definition, you bypass all characters until the *BRACES* symbol is encountered because certainly no function definition can occur between the *PARENS* and *BRACES* symbols. It only remains to append the function name to the output file, funcs.txt.

Finishing Off bldfuncs

The main function of *bldfuncs* opens *funcs.txt* and calls *get_names_one_file* for every source file specified on the input line. To enable expansion of wildcards on the command line, the module *setargv* (provided with the Microsoft C compiler) is linked in with *bldfuncs*. (See the *make* file in *bldfuncs* in Example 2, page 31.) This module lets you use a command such as:

```
bldfuncs *.c \othersource\*.c
```

which results in a *funcs.txt* file containing the names of functions in every C source file in both the current directory and the directory *\othersource*.

bldfuncs uses lots of stack space for local storage, so my *make* file specifies a stack size of 14000 in the link step:

```
link bldfuncs + \msc5\lib\setargv/
```

ST:14000/NOE;

Finally, Listing Two, page 74, is the file *bldfuncs.doc*, which contains a help page for *bldfuncs*.

The Second Half

Now let's look at *getf* (Listing Three, starting on page 75), whose job is to scan *funcs.txt* for a specified function name (or for all names matching a specified pattern) and to present the appropriate source file in the editor. As the first step in the process, *getf* looks for the DOS environment variable *GETFEDIT*, which must contain the control string used to construct the command line that invokes the editor. For example, if your editor were named *edit*, you would first issue the DOS command:

```
set GETFEDIT = edit %s
```

and if this line were placed in your autoexec.bat file, it would look like:

```
set GETFEDIT = edit %%s
```

Note the use of the %% sequence to avoid interpretation of the % symbol by command.com.

The Microsoft string function *strstr* is used to verify that there is at least one occurrence of %s in *GETFEDIT*. (Given two strings *s1* and *s2*, *strstr(s1,s2)* returns a character pointer to the first occurrence of *s2* in *s1* or *NULL* if there is none.) Then, the string function *strtok* is used to scan *GETFEDIT* for an initial token (delimited by white space) that should be the program name of the editor. Recall that given two strings *s* and *delim*, *strtok(s,delim)* scans *s* for a token ending with a character from the *delim* string, replaces this character with a null, and returns a character pointer to this token in *s*. Subsequent calls to *strtok* have the form *strtok(NULL,delim)* and return character pointers to subsequent tokens, finally returning *NULL* when no more tokens are to be had.

The program name of the editor is stored in *pgm_name*, and the remainder of the *GETFEDIT* string is stored in *arg1_ctl*; these are used later in constructing an *exec* call to invoke the editor.

Next, *getf* opens *funcs.txt* and scans for file names, which end with a colon. Any such name is saved in *file_token*. Having obtained *file_token*, *getf* scans for function names that match the pattern *func_name* (obtained from the *getf* command line) using the function *patn_match*. Any such match is stored in the arrays *func_choices* and (corresponding) *file_choices*, indexed by the integer *num_choices*.

When this scan terminates, *num_choices* is examined to see if any matches were found. If there was no match, an appropriate message is printed and the editor is not invoked. If there was exactly one match, the function *edit* is invoked to bring up the specified file and function in the editor. If several matches occurred, the function *ask_for_file* is invoked to prompt the user for a choice among these, and then the function *edit* is invoked as in the case of one match.

The function *patn_match( )* accepts a pattern and a string as arguments and returns *TRUE* or *FALSE* depending upon whether or not a match occurred. The pattern-matching rules are tailored to the case of function identifiers as follows: a *?* (question mark) matches any single character, an * (asterisk) matches the remainder of any string, and a % (percent sign) matches any string up to the next underscore character or the end of the string.

The function *edit( )* accepts a function and a file as parameters and uses the previously obtained strings *pgm_name* and *arg1_ctl* (parsed from the *GETFEDIT* variable) as parameters for an *execlp* call, which, if successful, will overlay the currently executing program, *getf*, with the editor. The *p* in *execlp* serves as a reminder that the DOS *PATH* variable is used to locate *pgm_name*.

The function *ask_for_file* uses the arrays *func_choices* and *file_choices* and the index variable *num_choices* to present the user with a menu of possible functions matching the pattern entered on the *getf* command line and the corresponding files in which they reside. If the command getf *get_%_data* were issued, a typical menu might look like:

```
Which one? (CR to exit)
 1: get_all_data in getdata.c
 2: get_good_data in valid.c
 3. get_some_data in input.c
Enter number:
```

When a listed number is chosen, the *edit* function is invoked using the specified elements of *func_choices* and *file_choices*.

A *make* file for *getf* is shown in Example 3, page 33. In order to avoid any stack-space problems, a generous allocation of 14,000 stack bytes is made in the link step:

```
link getf/ST:14000;
```

The stack allocation of 14,000 bytes is the same for both *bldfuncs* and *getf*.

Like *bldfuncs*, *getf* also has a short *help* file (see Listing Four, page 81).

Summing Up

I have found these two programs in conjunction to be an excellent time-saver whenever I have to go wandering through source code files and have been using them extensively since I wrote them. The programs have saved me a great deal of time, and fall into the performance category of "fast enough." Although I've no doubt that there are more efficient ways of parsing through C source code than my multiple-filter strategy, the method I chose had two outstanding advantages over other methods: it was easy to program, and it was simple to debug.

Of course, the techniques presented in this article could be generalized to languages other than C. *getf* would require almost no changes, but a new version of *bldfuncs* would be required in order to do the parsing specific to the language desired. I'd be interested in hearing from anyone who does the conversion to another language.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 70.)

AN AID TO DOCUMENTING C

The only thing more painful than debugging your program is documenting it. This utility takes some of the sting out of documenting C projects.

by Stewart Nutter

Did you ever get caught between a code deadline and the need for revised documentation? All too often, large programming projects reach completion with a large discrepancy between the design documentation and the final actual program. With C's ability to divide a big task into small tasks or modules, keeping the documentation up to date can be a headache. In addition, with the creation of each new module, the effort to revise the documentation can triple. In the crunch to get the program done on time, the master design documentation never seems to get updated.

Like many *DDJ* readers, I've faced this documentation problem myself. I recently wrote a large program in C that eventually had more than 750,000 bytes of source code. Not only were there more than 220 different source-code modules, but there were lots of assembly code modules and screen files as well. As you might guess, many of the modules weren't specified in the original design. This became a major problem, because I wrote the program for another company that planned to maintain it from my documentation. The way the various modules fit together was important, and the best way I could show the relationships was by drawing a tree structure. The problem I faced was that detailing over 220 modules is very time-consuming and, given the project deadline, I probably wouldn't finish.

Here is my solution: a utility to document the source-code modules. Like many programs, this one was based on a philosophy of laziness. My thinking has always been that to do something once is fine, but if the task is boring or repetitious, then it's time for a computer. It was obvious from the circumstances that I needed a program that would do the tedious tree charting for the project.

A quick look through magazines for exactly what I needed did not produce results. But after examining an ad for a program that showed the control flow of a single source-code module, I quickly realized that I could use the same concept for every module and function of a C project. After thinking about it, I came up with an outline of my utility program. The following list are features that I wanted in my charting program:

- It should take as input a list of filenames.
- It should output a list of all functions, in the order called, starting with *main*.
- The program should count the number of times each function is invoked.
- The program should maintain the module's filename that contains the function's source code.
- It should count the number of lines and characters in each module.
- It should identify the functions not used in the project.
- It should create a function index.

I eventually wound up writing a program to satisfy this list. The rest of this article details the operation of my program, which I call *cp* (for *C printer*).

Figure 1, on opposite page, shows a greatly trimmed printout from the program. In this case, I used the utility to document the various functions and modules in the source code for the *cp* program itself. Figure 1 gives an example of the first and largest part of any report: the module relationships. The generated report also contains statistics for the sizes of the modules, with function indexes including a list of functions not used.

Stewart Nutter is a senior programmer for York International Inc. He may be reached at 111 W. Gay St., Red Lion, PA 17356.

The cp program itself is broken into three files: cp.c (the main module), cpfuncts.c (containing the support functions), and cpheader.h (the header module). Listings One, Two, and Three begin on page 84 with cp.c, chheader.h, an cpfuncts.c, respectively.

How the Program Works

The cp program starts by scanning the module files contained in the source-list file for every function either called or defined. (The source-list file is simply an ASCII text file that contains the names of the modules used in the project, one name per line.) After the scan, the program stores all the gathered information in an array of module structures. Each entry in the module structure contains the following information:

- A pointer to the name of the module.
- A pointer to the name of the defined function.
- A pointer to a sequential list of different function structures that are called.
- The number of different functions that are called by the defined function.
- A pointer to a linked list of page references.
- The number of times the defined function is used in the program.

As the program reads each source file, the program keeps a structured list of module names, recording the number of lines and characters, for statistical purposes.

After finding a called function, the program adds the function's name to the sequential list of functions. The list is maintained in a first-come-first served order. Prior to placing a function in the list, the program checks for duplicate entries. If the function is already in the list, then the used count is incremented for the duplicate entry.

The program continues until it has read all the source files. It then sorts the list of defined functions by function name. The printout is started by searching the sorted list for the function *main*. If the function *main* was found, the program prints each function in the called function list. If a called function name is found in the defined module list, then the usage count is checked. All defined functions that are used only once are then recursively passed to the *print* function. When the program completely prints *main*, the print routine prints the defined functions that are used more than once. While each function is printed, a linked list of page references is updated.

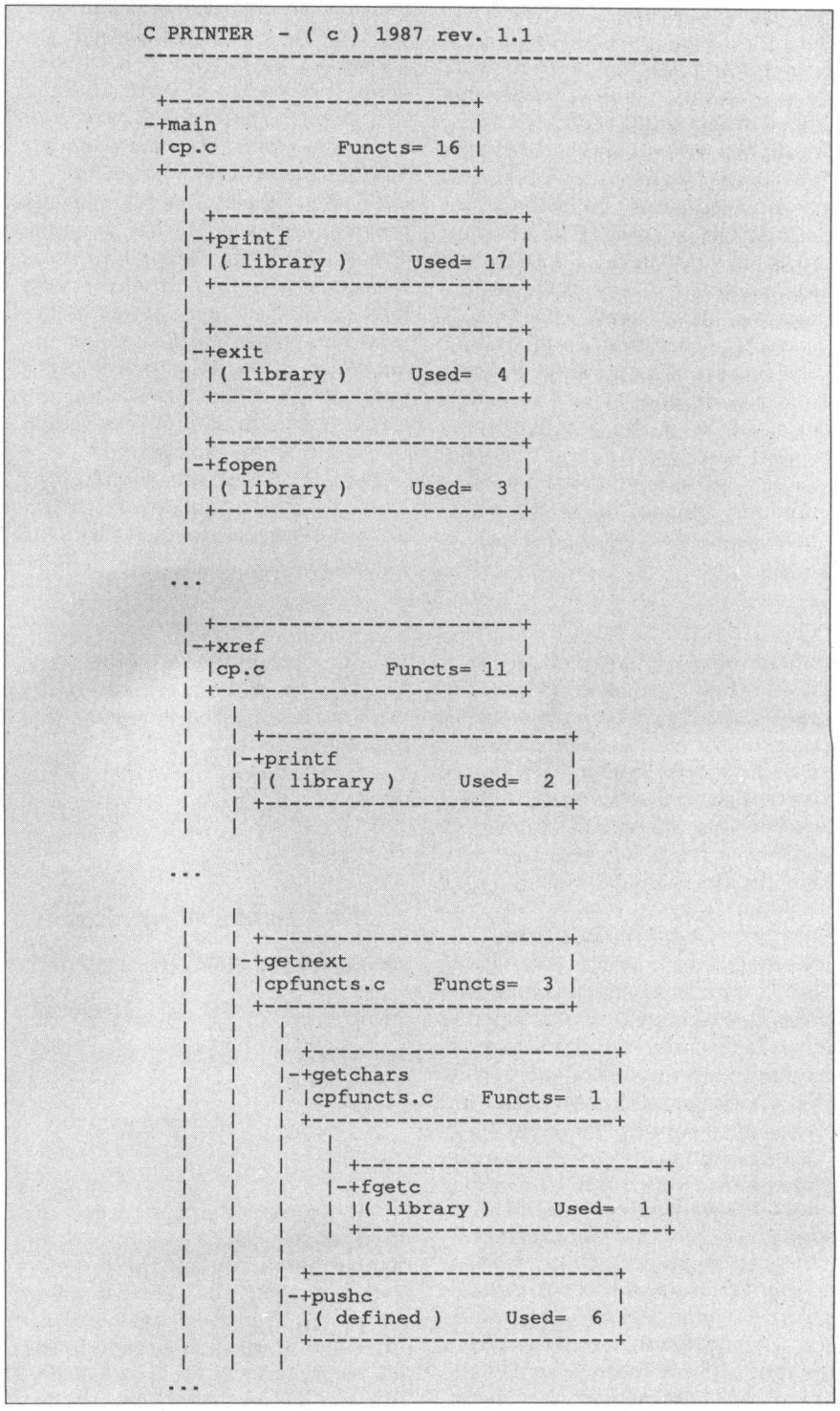


Figure 1: *Sample output from the C printer program. Following this function map come reports on the module sizes and function lists indexed by both module and page number.*

(continued from page 41)

As you can see in the sample printout, the program prints each function inside a text box. Printed in the lower-left corner of the box is a word or description about the function. If the function is a *root* function or a defining function, then the program prints the name of the module where the function is defined. If the function is defined somewhere in the list of code modules and is currently a called function, then the word defined is printed. The program prints the word library if the function is not defined in any of the source modules. The word *recursive* is printed if the function is calling either itself or one of its parent functions. The program checks for a recursion depth of 50. If the function being printed is a *root* function, then the program prints the number of called functions. Otherwise, the number of times each called function is used is printed.

Just the Stats, Madam

After printing all the functions, the program prints statistical data collected during the scan of the modules. The first list contains the name each module with its number of lines and bytes. At the end of the list is the total byte count and the total number of lines. The second list contains the names of all functions that were defined in the list of code modules. The functions are printed in ascending order, with the name of the module file where the function is defined and the number of times the function was used in the program. The third list is an index or a cross reference to the pages where each module is used or defined in the printout. The last list contains the names of all the functions that are not used or called in the program, along with the module where the unused function is defined.

The header file cpheader.h (Listing Two on page 87) contains the definitions of the structures and the global variables. Each source file that includes the header must define the constant *MAINMODULE*. If *MAINMODULE* equals zero, then the global variables are stated as external; if *MAINMODULE* does not equal zero, then the variables are declared. In the file that contains *main* the constant *MAINMODULE* is defined as 1 so that the global variables will be initially defined. The other files should define *MAINMODULE* as 0 so the variables will be externals. The constant *MAXFNCTS*, in the header file, is the total number of different called functions used in each function. The constant *MAXMODULES* is the total number of different files that make up the program.

The main file cp.c contains the *main* function, which checks for command-line inputs, opens the source-list file, displays error messages, and prints the data to the destination device. Each file is opened in *main* and the filename is passed to the function *xref*.

The function *xref* looks through the module file looking for functions. Functions to *xref* are text words followed by an open parenthesis. Defined functions occur when the open brace count is 0. When the open brace count is greater than 1, the function is a called function. Defined functions are differentiated from function declarations because a semicolon follows a function declaration.

Supporting Characters

The module file cpfuncts.c (Listing Three on page 87) contains functions called by *main* and *xref*. The function *getnext* processes each character from the source-module file, and rejects any character between quotation marks, apostrophes, or comments.

The function *getchars* either reads a character from the file or from a last-in-first-out buffer. The buffer contains characters that were read once and need to be reread. If there are no more characters to read, the program returns an end-of-file character.

If a character that was read from the source file needs to be reread, the function *pushc* will place the character into a last-in-first-out buffer. The function checks for room in the buffer prior to putting the character into the buffer.

The print program starts by scanning the module files contained in the source list.

The function *addlist* adds the name of the called function to the defined function's list. The function checks to see if the called function is one of C's reserved keywords or if the called function is already in the list before adding it. If the called function already exists in the list, then *addlist* increments the usage count.

The function *find_mod* searches the sorted list of defined functions for the supplied function name. The function *find_mod* returns the index number of the defined function if it is in the list; otherwise, *find_mode* returns -1 to indicate that it was not found.

The function *doprint* starts the printout with a defined function's index. This function is the "meat and potatoes" of the printer program: *doprint* prints all the called functions of the supplied function index. If a called function is defined only once in the source files, then *doprint* calls itself with the new defined function index. It also knows whether the function is defined, a library function, or a recursive call. Each time that *doprint* is called, it checks a recursion list. If the new function is not in the list, it is then added to the end of the list. When *doprint* exits, it removes the last entered item from the recursion list.

The function *getstats* checks for the use of each defined function in the source files. The function *getstats* scans the list of used functions collected by the function *xref*. When each used function is found in the defined function list, (*find_mod*) the used count for the defined function is incremented.

The function *pagebreak* checks the count of printed lines to see if enough lines have been printed to print both a formfeed and a new page header. The function forces the printing of a new page by setting the line value to 99, which is greater than the number of lines allowed per page.

The function *recur_chk* checks through a list of parent functions for the current function name. If the function is in the list, then the current function is a recursive function. The recursion list is a first-in-last-out list of function names.

The function *setpage* adds the page number to the linked list of structures containing page data for the supplied function name. The last entry in the list contains a NULL pointer to the next entry.

The function *strcheck* checks a character set for valid characters in a function name. If the character is a valid function-name character, *strcheck* returns a value of 1; if not, *strcheck* returns a value of 0.

Construction Details

I used Microsoft's C compiler version 4.0 and its *make* utility program. I then compiled the program using the large memory model. As far as I can tell, there aren't any dangerous dependencies on Microsoft C in the code, so translating it to another version of C should be straightforward.

I controlled the project with Microsoft's *make* utility; the *make* file is shown in Example 1, on page 48. (If you don't have *make* then you'll have to enter the appropriate command lines yourself.) The *make* utility compares the dates of the independent files, which are to the right of the colon, to the dependent file, which is left of the colon. If any of the independent files are newer than the dependent file—implying that you've edited one of them—then the program executes the list of commands that follows. To link the resulting object files, type on the command line

```
link cp + cpfuncts;
```

To use the program, you must create a file that contains a list of all your C source filenames. You do not need to sort the file: cp sorts the file itself so it can tell how far along it is during the processing. For example, the program cp.exe is created from the two source files cp.c and cpfuncts.c. Example 2, on page 48, shows the contents of the list file. The command line in this example should have the following information: *cp mylist outfile*. *mylist* is the file that contains the names of the source-code files. *outfile* is the name of the output file or device. If the cp program is called without any arguments on the command line then it will display the proper usage for the program. The program defaults to an output file name of prn if no file name is given on the command line. To invoke cp for the example program, type

```
cp cplist cplist.prn <Enter>
```

The program cp will read the file cplist for the list of filenames and will output the printout to the file cplist.prn.

Revisions and Improvements

It's difficult to resist the urge to improve a program while you're actually programming, and it's even harder once you've started testing the results. I found this program to be no different. Since I finished the first version of cp, I've added a number of improvements, including these:

1. Support for Microsoft Windows C programs.
2. The ability to enter a starting function name, instead of just *main* from the command line. For example, in a windows program, *main* would be named *WinMain*.
3. Command-line arguments for page width and length.
4. The saving of the first comment of each module to be printed as a short module description list.
5. A command-line switch to print the statistical information only.

Feel free to use the program at home or in your business, but please don't distribute it for profit. Programs like this are tools to increase your productivity and should be freely shared. Please direct any questions, comments, improvements, or bugs to the address at the beginning of the article. If you need a reply, include a self-addressed stamped envelope.

```
cp.obj : cp.c cpheader.h
   cl -Ox -AL cp.c -c

cpfuncts.obj : cpfuncts.c cpheader.h
   cl -AL -Ox cpfuncts.c -c

cp.exe : cp.obj cpfuncts.obj
   link cp+cpfuncts;
```

Example 1: *A sample make file for cp, specifying large model and maximum optimization.*

```
cp.c
cpfuncts.c
```

Example 2: *Sample file list created for cp.c.*

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 84.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

A TOOL FOR SECRET KEY CRYPTOGRAPHY

Software rotor techniques are not constrained by the limitations imposed by mechanical gearing on electro-mechanical machines.

John R. Michener

Many of the proprietary algorithms used in commercial software packages have proven to be vulnerable to the sophisticated statistical tools of cryptanalysis. As you might expect, this creates havoc and even panic in those environments (banks, defense contractors, and so on) where security is a necessity. With this in mind, this article will discuss a cryptographic algorithm that can be used to construct secret key cryptographic systems that are relatively resistant to analysis.

Before discussing crypto-algorithms, however, a few words should be said about how codes are broken. Cryptanalysis is a form of applied mathematics and statistics in which the cryptanalyst attempts to strip from the underlying message the confusion supplied by the cryptographic process. The basic tool of the cryptographer is a thorough knowledge of the language and its properties. Written English typically has a redundancy on the order of 75 percent (when using a 26 character alphabet), resulting in each character of English text transmitting somewhat over 1 bit of information. Thi__ se__ten__e i__an__exa__ple__of__his __eff__ct __eca__se __ne __n fo__r c__ara__ter__ ha__ be__n d__let__d, __nd __epl__ced__by __ __. It is not very difficult to determine that the text of the previous sentence is "This sentence is an example of this effect because one in four characters has been deleted and replaced by a __." English text encoded as ASCII characters has a redundancy on the order of 85 percent (8 bits per character, resulting in an alphabet of 256 characters). In fact, many individuals enjoy such language reconstruction problems to the extent that they spend substantial portions of their own time solving similar problems in the form of acrostics and crossword puzzles.

Many documents may be viewed as having far lower information contents than the 1+ bit per character estimate. If we look at "boilerplate" phrases (such as we always find in the fine print on car loan, credit card, or mortgage agreements) we find that such phrases tend to be highly standardized within any given field. Rather than sending the phrase or paragraph, it would, in principle, be sufficient to send two pieces of information denoting the specialty area and the phrase identifier. Typically, only a few bytes would be necessary to represent an entire paragraph.

Computer language files are frequently far more structured and predictable than documents written in English. This is particularly true for highly structured transactions where it must be assumed that the attacker has detailed knowledge of the message structure. An example of such a situation is the communications of bank money transfer machines with their host mainframes. A likely attacker of such a system is a bank employee who has detailed knowledge of the protocols used, but does not have the key to encipher the communications. The initiating machine must provide the user's identification number, PIN number, the requested transaction, and authentication information to the host computer in such a manner that an attacker is unable to extract money from the system.

Predictability (including the message structure) and the redundancy

John Michener is a senior research scientist. He is working in the field of simulation and beam probing. He can be reached at 177 Moore St., Princeton, NJ 08540

of source material provide cryptanalysts with their starting material as well as checks for the accuracy of their assumptions. Compression of the data allows a substantial reduction in the message redundancy and length, as well as largely destroying the message structure. Encryption of the compressed message results in ciphertext with a far higher resistance to cryptanalysis than ciphertext of an uncompressed message. It should be noted that the output of any encryption system appears to be quite random. Because enciphered text can not be compressed, text must be compressed before it is enciphered.

Both the history of cryptographic security and common sense tell us that any builder of cryptographic systems must expect that attackers will have a detailed knowledge of the cryptographic system itself. One must also assume that the attackers will have large amounts of matched ciphertext and plaintext available for analysis (this is called a known plaintext attack). No cryptographic system should be considered for use if it cannot resist such an attack. The builder of the cryptographic system should make every effort to minimize the attacker's knowledge of the message structure. One of the best ways of doing this is to reduce the redundancy of the message before it is encrypted.

Data Compaction

The scheme taken to compact the source material is typically dependent upon the material to be compressed, but it may well involve some form of Huffman encoding at the character or word level, the use of dictionary compaction schemes, codebook techniques for common phrases, general purpose compaction techniques, or some combination of these.

Huffman encoding, one of the better known compression techniques, commonly encodes ASCII text into a character set with a variable number of bits. The most frequent characters are assigned to the shortest encoded representations. Huffman encoding at the character level typically reduces English text file lengths by 30 percent. Because of variable length characters, a single byte in Huffman encoding may represent from less than one to parts of three characters.

Well designed compaction algorithms can reduce the number of bits necessary to encode a character in arbitrary English text to between 2 and 3 bits. Compaction algorithms for highly structured messages can and should be optimized to remove the message structure and redundancy in order to maximize the difficulties faced by the cryptanalysist. When transposition operations are applied to compacted text, structure needed for the decompression routines is diffused over the message, further hindering the cryptanalysist, who must both decipher the encryption scheme and reconstruct the compressed text.

A Generalized Rotor Cryptographic Operator

In the 1930s encoding machines were introduced that utilized a series of rotors: electrical scrambling units that substitute one character for another. The shaft-mounted rotors were turned by gears and cams. Encryption keys were established by changing rotors, their order in the machine, rotational settings, and other mechanical parameters, thus making analysis of the resulting code difficult. However, the relatively regular manner in which the rotors moved in relationship to one another, as well as the fixed order of the rotors during any given encipherment session, rendered electro-mechanical rotor machines vulnerable to analysis.

Software implementations of rotor techniques are not constrained by the limitations imposed by mechanical gearing on the electro-mechanical machines. A linked substitution list (rotor) technique is resistant to analysis and is well suited for implementation on microcomputers. I call this technique the "Generalized Rotor." Its only drawback is the requirement to store the rotor sets in memory during processing and in a readable medium between processing sessions.

This technique utilizes a set of substitution lists (rotors) such that each rotor is a randomly scrambled list of numbers from 0 to p-1, where p is the list length (rotor size), with each element appearing only once. The decryption rotors are the inverse of the encryption rotors, and are calculated from them, but need to be available as a separate rotor set. The code in Listing One, on page 96, provides a means of generating a set of encryption and decryption rotors given a sufficiently long stream of random bytes. No relationships or correlations should exist between the various rotors in the rotor set or between elements in any given rotor. In the descriptions that follow, p and the number of rotors in the rotor set are 256 because of the calculational convenience of this choice. Other choices could easily be made. If ASCII files are to be transformed and it is desired, from an implementation point of view, to keep the rotors and other working variables within a 64K block, it would be logical to use 128 rotors, each 128 bytes long.

The generalized rotor operates by using the outputs of pseudo-random sequence generators to choose the rotors, as well as the relative offsets of each rotor in the working set. After each character is enciphered, the rotors in the working set and their offsets are set to the new values supplied by the pseudo-random sequence generators. The total change of the rotor set and offset values after each character is processed eliminates the regularities that render mechanical rotor systems vulnerable to analysis. There are several "keys" in this system: the contents of the encryption and decryption rotors, the algorithms used in the pseudo-random sequence generators, and their starting contents. The seed of each pseudo-random generator, or the string from which it is created, is the working key for the system and should be changed frequently. The rotor contents and pseudo-random sequence generator algorithms constitute the fixed key and can be changed at greater intervals. The C code block in Listing Two, on page 96, implements encipherment and decipherment of individual characters utilizing generalized rotor substitution operations where the rotors are stored as two dimensional arrays. Use of these routines requires that the message stream be fed through the transformations

character by character.

The generalized rotor operator may be viewed as a very complex mixer of multiple pseudo-random sequences with a chosen text stream. The mixing of both is resistant to analysis and is easy to implement. The encryption and decryption rotor sets are implemented as large single arrays, minimizing the index computations necessary to perform the operations. Each rotor uses two pseudo-random values, one for the rotor selection and the other for the rotor offset. The most secure way of selecting these values is to produce each one by a separate generator. The generators have periods that are long and do not share common factors, preventing undesired periodicities in the sequences. A four rotor system thus needs eight sequence generators. The additive generators discussed earlier are well suited for such sequences.

No cryptographic system should be considered for use if it cannot resist such an attack.

The use of generalized rotor operations—rather than addition—as the combining operation in such a feedback shift register is fast and hard to analyze. If the two values fed back determine the rotor choice and rotor offset values, the resulting value is derived from the rotor transformation X(n) = rot[X(n-a)][X(n-b)]. Such a combining operation has apparently random mapping characteristics and is not easily inverted. The inverse operation of determining what prior states could have yielded a given output is many to one: for rotor sets of 256 rotors, each 256 long, there are 256 previous states that could have yielded a given output.

There is no theory to predict the period lengths of the sequences produced by such generators. The apparently random nature of the mappings suggests that the period is likely to be on the order of p^ -1, but can not be guaranteed to be this large. Such sequence generators are very fast due to the speed of memory access, and they add considerable difficulties to the work of the cryptanalysist. This is the whole point of encryption.

Reference Notes

Martin Kochanski, "A Survey of Data Insecurity Packages," Cryptologia (Jan. 1987): 1–15.

Rivest, Shamir, and Adleman "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," Proceedings of the IEEE (March 1979): 397–426.

Martin E. Hellman, "The Mathematics of Public Key Cryptography," *Scientific American* (Aug. 1979): 146.

Martin E. Hellman and W. Diffie, "New Directions in Cryptography," IEEE Transactions on Information Theory (Nov. 1976): 644–654.

W. Diffie and Martin E. Hellman, "Privacy and Authentication, an Introduction to Cryptography," Proceedings of the IEEE (March 1977): 397–426.

David Kahn, *The Codebreakers* (New York, N.Y.: MacMillan Co., 1967).

Rudolph F. Lauer, *Computer Simulation of Classical Substitution Cryptographic Systems*, (Laguna Hills, Calif.: Aegean Park Press, 1981). Gilbert Held, *Data Compression* (New York, N.Y.: John Wiley & Sons, 1983).

Frank Rubin, "Experiments in Text File Compression," Communications of the ACM (Nov. 1976): 617–623.

Eugene S. Schwartz, "A Dictionary for Minimum Redundancy Encoding," Journal of the ACM (Oct. 1963): vol. 10, no. 4, 423–439.

Richard Tropper, "Binary Coded Text," *Byte* (April 1982): 398–413.

Jorma Rissanen, "A Universal Data Compression System," IEEE Transactions on Information Theory (Sept. 1983): 656–664.

J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable Rate Coding," IEEE Transactions on Information Theory (Sept. 1978): 530–536.

Bruce Hahn, "A New Technique for

Compression and Storage of Data," Communications of the ACM (Aug. 1974): 434–436.
Terry A. Welch, "A Technique for High Performance Data Compression," IEEE Computer (June 1984): 8–19.
Ian H. Witten, Neal M. Radford, and John G. Cleary, "Arithmetic Coding for Data Compression," Communications of the ACM (June 1987): 520–540.
John G. Cleary and Ian H. Whitten, "Data Compression Using Adaptive Coding and Partial String Matching," IEEE Transactions Communication (April 1984): 396–402.
Donald E. Knuth, *The Art of Computer Programming*, vol.1, Seminumerical Algorithms, 2d ed, (Reading, Mass.: Addison-Wesley 1981).
Solomon W. Golomb, *Shift Register Sequences*, Revised Second Edition, (Laguna Hills, Calif. Aegean Park Press, 1982).
Henry Becker and Fredrick Piper, *Cipher Systems*, (New York, N.Y.: John Wiley & Sons, 1982).
T. Siegentahler, "Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications," IEEE Transactions on Information Theory (Sept. 1984): 776–780.
Herbert S. Bright and Richard L. Enison, "Quasi-Random Number Sequences From a Long-Period TLP Generator With Remarks on Application to Cryptography," Computing Surveys (Dec. 1979): 357–370.
T. Etzion and A. Lempel, "Algorithms For The Generation of Full-Length Shift-Register Sequences," IEEE Transactions on Information Theory (May 1984): 480–484.
Ronald L. Rivest, "Forwards and Backwards Encryption," Cryptologia (Jan. 1980): 30–34.
John R. Michener, "The 'Generalized Rotor' Cryptographic Operator and Some of its Applications," Cryptologia (April 1985): 97–114.
John R. Michener, "Application of the Generalized Rotor Cryptographic Operator in the Construction of Substitution-Permutation Network Block Codes," Cryptologia (July 1985): 193–202.
John R. Michener, "The Application of Key Dependent and Variable Rotor Sets to Generalized Rotor Cryptographic Systems," Cryptologia, (July 1987): 166–171.
Claude E. Shannon, "Communication Theory of Secrecy Systems," Bell System Technical Journal (Oct. 1949): 656–715.
John Kam and George Davida, "Structured Design of Substitution-Permutation Encryption Networks," IEEE Transactions on Computers (Oct. 1979): 747–753.
Horst Feistel, "Cryptography and Computer Privacy," *Scientific American* (May 1973): 15–23.
Arthur Sorkin, "Lucifer, A Cryptographic Algorithm," Cryptologia (Jan. 1984): 24–41.
Dov Andelman and James Reeds, "On the Cryptanalysis of Rotor Machines and Substitution-Permutation Networks," IEEE Transactions on Information Theory (July 1982): 578–584.
John R. Michener, "The Use of Complete, Nonlinear, Block Codes for Nonlinear, Noninvertible Mixing of Pseudo-Random Sequences," Cryptologia (April 1987): 108–112.
M.B. Greenlee, "Requirements for Key Management Protocols in the Wholesale Financial Services Industry," IEEE Communications Magazine (Sept. 1985): 22–28.
David M. Baleson, "Automated Distribution of Cryptographic Keys Using the Financial Institute Key Management Standard," IEEE Communications Magazine (Sept. 1985): 41–46.
Christian Mueller-Schloer, "A Microprocessor-Based Cryptoprocessor," IEEE Micro (Oct. 1983): 5–15.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 96.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

OBJECTIVE-C

Just as functional hardware modules can be reused, so can classes in a software design.

by Ernest R. Tello

One good indication that object-oriented programming is here to stay would be the appearance of some serious delivery environments for applications developed with this programming paradigm. So far, Objective-C from Stepstone (formerly PPI) looks the most promising of the tools in this category. What I find interesting here is that this system was developed by C programmers who were looking primarily at the advantages object-oriented programming affords for handling conventional programming projects. In spite of this emphasis on the part of its developers, I believe that Objective-C may hold some promise as a delivery environment for AI applications as well.

The release of Objective-C that I used for my evaluation was the Version 3.31 implementation ported to the IBM PC AT. The language can also run on VAXes, Sun workstations, and the HP-9000 series. The AT version of Objective-C comes on one high-density disk and includes the compiler, libraries, and the main classes also in source form. At this point, the only C compiler on PCs supported is Version 4.0 of the Microsoft C compiler; support for Version 5.0 is planned.

Version 3.31 represents a relatively mature implementation of Objective-C that reflects a year or two's experience with problems that programmers have encountered. Because of difficulties that arose with some of the classes in the foundation library, although they have been included for the benefit of those who have already written code that uses them, the manual warns about the problems and suggests that they not be used. The classes in this category are the *BytArray* and *Bag* classes.

The Objective-C compiler consists of two executable files—the driver program, objcc.exe, and the actual Objective-C program itself, objc.exe. The driver program first calls the Microsoft cl.exe program to check syntax and then calls the Objective-C compiler. There is no need to specify libraries at link time, except if their paths cannot be found. The library references are embedded in the .obj files by Objective-C.

One of the things that puts Objective-C potentially in the category of a suitable delivery environment for AI applications is its feature of dynamic run-time binding for all objects. It is able to accomplish this even though it compiles its own code into C for subsequent compilation by a C compiler.

An important difference between Objective-C and other object-oriented systems such as Smalltalk is that it is really a hybrid language. So, just as with object-oriented Lisp systems, programmers always have the option of writing code in conventional C. Another important difference between Objective-C and Smalltalk is the difference in the size of the class libraries. Smalltalk-80 comes with a substantial amount of code available for reuse in source code form. Objective-C, although it offers considerably more in this department than does C++, lags substantially behind Smalltalk, which is the senior member among object-oriented languages.

In certain respects, Objective-C is a rather conservative object-oriented language—that is, there are no substantial innovations in object-oriented concepts here that were not already in Smalltalk many years ago. On the other hand, not every powerful feature of Smalltalk is found in Objective-C either. If it can be considered as a hybrid of C and Smalltalk, then the more dominant parent by far is clearly C. Looking at it from the perspective of C, Objective-C has extended the C language by adding one new data type, *Object*, and one new operation, message expression.

Classes

The syntax of Objective-C is, for the most part, quite straightforward. To declare a new class, you use the equal sign (=), and you use the colon (:) to declare its superclass. Other items in the class definition are set

Ernest R. Tello is an AI consultant, lecturer, and writer of publications in the field of artificial intelligence. His latest book is titled Mastering AI Tools And Techniques. *He can be reached at 1518 W. Cliff Dr., Santa Cruz, CA 95060.*

off by parentheses, and all data declarations are set off in curly braces. So, for example, the following expression:

```
= Array:Object { short capacity; }
```

declares *Array* as a subclass of *Object* with the instance variable *capacity* declared as a *short* integer.

There are two types of methods—class methods and instance methods—just as in Smalltalk, and they are defined using the plus (+) and minus (-) signs, respectively. Instances are usually created, as in Smalltalk, by sending the message *new* to the parent class.

Objective-C has two main types of message expressions—unary expressions and keyword expressions. There are no binary expressions like those used in Smalltalk. Hence, the expression:

```
id myarray = [ByteArray new:80];
```

creates a new instance of the class *ByteArray* sized at 80 units, and the definition of the method *new* for the *Object* class is just:

```
+ new { return (*__alloc)(self, 0); }
```

Here a built-in primitive is called on to do the job of allocating memory for an object. No further work is needed because the object is the simplest possible abstraction class.

On the other hand, the *new:* (pronounced "new colon") message for the *Array* class has this high-level Objective-C definition:

```
+ new:(int)nElements {
    self = (*__alloc)(self, nElements*
                      [self ndxVarSize]);
    capacity = nElements; return self;
```

All messages in Objective-C are set between square brackets, so the expression *[self ndxVarSize]* is a message that the receiver object will be sent. The *ndxVarSize* message is an *Array* class method that is redefined:

```
+ (int)ndxVarSize { return (int)
        [self subclassResponsibility]; }
```

The *subclassResponsibility* method simply prints the message "Subclass should override this message" when called. The expression *capacity = nElements* simply sets the capacity of the array to whatever argument is suppled to *new:*.

Example 1, this page, shows the entire class library lists of the two versions of Objective-C as they might appear in a Smalltalk system browser, if such a thing existed in Objective-C.

In Objective-C classes are also referred to as factory objects to underscore that a class is an object in its own right whose main function is to serve as a template for the creation of instances and subclasses. As you can see, however, there are no *Class* and *Metaclass* classes present. Classes in Objective-C are not instances of a metaclass object and are not created by sending messages to a metaclass, as is true in Smalltalk, Xerox LOOPS, and many object-oriented Lisps. In itself, of course, this is not necessarily a bad thing and is simply another way of saying that Objective-C is a hybrid rather than a pure object-oriented language.

The library classes in Objective-C are arranged in roughly four broad categories: foundation classes; collection classes; other data-type classes such as *String;* and screen I/O classes. As you can see from the class hierarchy tables, in the new version of Objective-C, the AVL classes have been omitted but several others have been added. The *BalNode* abstract class and its subclass *SortCltn* are now used instead. *BalNode* is generic code capable of supporting implementations of any binary tree. *SortCltn*, a class that handles sorted collections, is the replacement for *AVLTree*.

Another important change is that the *Sequence* class is now implemented as a subclass of *IPSequence*. The latter implements sequencing quickly through any kind of collection by running in place over its contents. In order to accommodate the technique used by the *IPSequence* class, the *contents* method was added to the collection classes. It returns the pointer to the instance of *IdArray* that the receiver is using to store the members of its collection.

The *AsciiFiler* class is a new class that gathers all the file operations and is able to support the transfer of source files between machines of different architectures on the same network.

Another interesting new class added is *ObjGraph*, which is used to create a graph of a class hierarchy, meaning that of all the classes from which it inherits. The manual provides an example of the use of *ObjGraph* by implementing a method called *broadcast* that takes a method name or selector as an argument and sends this method to all the objects that can be reached by the receiver. To accommodate this new way of creating graphs, the *asGraph* method of the *Object* class has been rewritten.

Earlier I demonstrated some rudimentary operations with the *Array* class. Arrays are implemented differently in Objective-C from the way they are in other object-oriented languages. In Smalltalk, the *Array* class is a descendant of the collection

```
Object                                  Object
  Array                                   Array
    ByteArray                               BytArray
    IdArray                                 IdArray
    IntArray                                IntArray
  Assoc    (association)                  Assoc
    Cltn   (collection)                     Cltn
      OrdCltn   (ordered collection)          OrdCltn
      Set                                     Set
        Bag                                     Bag
    Dictionary                              Dictionary
      Stack                                   Stack
    AVLDict   (sorted dictionary)         AsciiFiler
  AVLTree                                 BalNode
  Point                                     SortCltn
  Rectangle                               IPSequence
  Sequence                                  Sequence
  String                                  ObjGraph
  Unknown                                 Point
                                          Rectangle
                                          String
                                          Unknown
```

Example 1: *Class library lists*

class, though not a direct descendant. In Objective-C, *Array* is a formal or abstract class that is the direct descendant of *Object*, the root class. This is obviously for efficiency purposes because C already has an implementation of arrays.

What is new with the *Array* class is the implementation of indexed instance variables instead of named instance variables. Arrays are fixed in size. Unlike the more sophisticated collection classes I will discuss later on, they cannot be increased in size when the number of elements reaches the maximum that was defined for a given array. The subclasses of *Array* handle arrays comprising the various C data types. As with most object-oriented languages, the *Array* class does not provide a facility for defining the dimension of arrays. To create multidimensional arrays, special subclasses of *Array* must be defined.

As you have seen, *Array* classes in Objective-C have a fixed capacity. Once an array instance of a certain capacity has been created, its size cannot be changed. This is not true, however, of collection classes, which are designed as "growable" classes that can later have more elements than specified by their initial capacity. The method that allows this in the Cltn class, expand, is written:

```
-expand { contents = [contents ca
      pacity: capacity + = capacity];
   return self;
   }
```

This is a transparent, high-level, Objective-C method definition that resets the value of the *contents* variable and uses a simple increment operation to double its capacity.

Objects in Objective-C are designed to reside in a single address space and to be identified exclusively by this address in system memory. This means that systems cannot generally be built with Objective-C when objects need to reside on disk or at other locations on a network. All objects in Objective-C have to reside in the host computer's memory.

Because Objective-C is a hybrid language, it allows "cheating." This means that, unlike a pure object-oriented language such as Smalltalk, it allows access to the protected memory of objects with C code that can access that memory directly. Needless to say, this is a good way to get into some deep trouble, defeating the whole idea of the encapsulation that is one of the main points of an object-oriented system, unless a programmer understands the implications of such actions fully. But the main gambit of a system such as Objective-C is to take that risk for the sake of greater performance.

Collection Data Structures

As with Smalltalk and most other object-oriented languages, the centerpiece of Objective-C for creating data structures is the group of collection classes. This part of the hierarchy has the following member classes:

```
Cltn
   OrdCltn
   Stack
   Set
      Dictionary
      Bag
BalNode
   SortCltn
```

The *Cltn* class is an abstract or formal class whose variables and methods are there to be inherited by its various descendants, which do all the actual work in programs. It is only the subclasses of *Cltn* that are meant actually to have instances.

The methods for collections are divided into about a dozen categories: instance creation, adding, removing, sequencing, elements perform, conversion, printing, freeing, copying, interrogation, comparison, and private methods. To understand them it is necessary to know a little about how collections work. The collection data structures are not used to store the elements themselves but pointers to the instances of the *IdArray* class that actually holds the members. The interrogation methods can be used in an application to query collections much as database queries, and searches are performed in conventional programming systems. So, for example, the *find* method searches for objects by name and returns them if they are present.

The *elementsPerform* methods are those that can map operations onto each element of a collection in turn. They do this by actually sending a message to each of the objects that are elements of the collection. One complexity is that different methods require different numbers of arguments, and Objective-C does not support functions with an optional number of arguments. The solution is that different *elementsPerform* methods must be implemented that accept different numbers of arguments. Versions are supplied that support up to three arguments; for more than this, it is no great problem to use these methods for implementing those that can accept more than three arguments.

As the name reveals, ordered collections are those whose elements are kept in order. Often more specialized subclasses of the OrdCltn class are used for handling queues and stacks. In ordered collections no nil entries are permitted, so whenever elements are removed from ordered collections, their contents are automatically compressed to take up the space created by the vacated element. Because of the nature of ordered collections, methods for adding elements to them specify a location at which to add them. These methods include *addFirst*, *addLast*, *insert:before*, and *insert:after*, which do the operations you would expect them to.

Stack implements collections that can keep entries in last-in, first-out order. In addition to the *push:* and *pop:* methods for accessing the contents of a stack, Objective-C stacks provide the *at:* and *removeAt:* methods for random access of stack elements. These random-access methods were new with Release 3.3. *Stack* manipulation methods include those that modify the order of stack elements, such as *swap*, as well as those such as *topElement* and *lastElement*, which provide information without making any modification to the stack.

Sets are collections that are only permitted to have one of each element. No duplicate elements are allowed. One application of sets that is particularly efficient is the creation of symbol tables. The *Set* class is implemented so that sets may contain any type of object. Several different types of object may even be collected in the same set. This means that, to add new methods to a set, an exhaustive search must be made

(continued from page 60)

of all existing elements in it.

The *Set* class in Objective-C supports a hashing facility so that, with the *hash* message, a set will place all objects it contains in a hash table for increased efficiency. An important limitation of sets as they are designed here, though, is that they are not designed to be changed dynamically. If the objects in an Objective-C set are modified, then the accessing facilities will no longer work correctly. This naturally limits their usefulness for AI applications.

The *Dictionary* class is a descendant of *Set* because dictionaries are implemented as a set of associations. In this case, you want to allow duplicate values, but each of the keys has to be unique. This is done by designing dictionaries to have a close relationship with the *Assoc* class. Associations store links between keys and values in such a way that these pairs can be stored in dictionaries and accessed by the key. Associations perform comparisons and equality testing by passing on messages to key objects. In addition to the methods inherited from *Set* and the other ancestor classes, *Dictionary* implements six new messages of its own: the class method *with:* for initializing new dictionaries and five methods for indexed access—*atKey:*, *atKey:put:*, *values*, *includesAssociation:*, and *includesKey:*.

As you saw earlier, the *SortCltn* class replaces the earlier *AVLTree* class in Objective-C. A sorted collection is one whose member elements always remain sorted. When an object is added, it must be inserted in the appropriate place right away. How the elements are ordered depends on what has been chosen as the value of the *cmpSel* instance variable. The default is the *compare* selector. Other options for its value are *invertCompare* and *dictCompare*, which are the names of methods. *Compare* and *invertCompare* are implemented in the Object or root class. The *dictCompare* method is implemented in the *String* class. Although *SortCltn* is not a subclass of *Cltn*, it acts as though it were. Its defined operations are "plug compatible" with it, according to the manual.

Another instance variable that alters the behavior of instances of the *SortCltn* class is the *addDupAction* variable. It can take on four different values: *ADD*, *REJECT*, *MERGE*, or *REPLACE*. These different options select different ways that duplicate elements are handled. If the *ADD* value is chosen, duplicates are permitted. In order to preserve the sorted ordering, any duplicate elements must always be the immediate successors of the elements they duplicate. With the *REJECT* option, duplicates are forbidden. As the name suggests, the *MERGE* option specifies that any duplicates will be merged using the *merge* method of the member's class.

```
- DemoPoint : Object ( Practice) {int xLoc,yLoc; }

+ create {return [ [ super new ] initialize] ; }

- initialize { xLoc = 100; yLoc = 100; return self; }

- print { printf ("This point's coordinates are (%d@%d ) \n" , xLoc, yLoc) ; }
=:
```

Example 2: *Short Objective-C demo program*

Graphics Classes

Objective-C provides the two rudimentary "graphics" classes—*Point* and *Rectangle*. As such, this is a far cry from a full object-oriented graphics system, but the construction of these classes is still quite informative. There are two main instance variables for a *Point*—*xLoc* and *yLoc*. Instance methods include all those it inherits from the *Object* class as well as those for setting and accessing the values of the two coordinates, those for moving the coordinate, and even those for performing simple math operations on the coordinate values.

To make use of instances of the *Point* class for making actual drawings, object-oriented systems generally have something like the *Pen* class used in Smalltalk and Actor, which implements the basic turtle graphics functions. In these systems, *Pen* is a descendant of the *BitBlt* class, which implements bit-block transfers. These classes do not come with the Objective-C system, so to use the *Point* and *Rectangle* classes, the equivalents of *BitBlt* and *Pen* would have to be implemented.

Example 2, shows a short demo function in Objective-C, and Example 3, below, shows the actual listing in C generated by the compiler. The C output has been reformatted in a less compressed form for easier reading.

The Vici Interpreter

An important accompaniment to the Objective-C compiler running on larger machines such as the Sun and VAX is an interpreter that understands both C and Objective-C. As even a superficial acquaintance with Smalltalk demonstrates, a dynamically interpreted environment is an ideal accompaniment to an object-oriented system. The interpreter evaluates and executes both statements typed in interactively and those read in from disk files. Perhaps best of all, not only can Vici be used stand-alone for development but it can also be linked into applications and become an integral part of them.

As with most interpreters, Vici allows you to inspect the value of any variable by typing its name. A trace facility for both C and Objective-C is built-in that provides various trace options, including tracing execution of compiled messages, tracing interpreted messages, and tracing the allocation of space to new objects. Although Vici allows statements to be entered interactively, it is not set up with a full-fledged built-in editor for serious programming. Typically, programmers will use an outside editor of their own choice in conjunction with Vici.

Symbolic Debugging

One convenient feature of the PC AT version of Objective-C is that you can use the Microsoft CodeView debugger as a source-level debugger for the Objective-C syntax. To be able to work with Objective-C source in CodeView, it is necessary to use the -g option initially when compiling the

application. Once done, you can bring up CodeView with Objective-C source displayed and can set breakpoints in the source for interrupting execution, enter expressions for evaluation, and so on.

You cannot inspect the values of objects directly, however. To inspect them, you must obtain access on the lower level by first using CodeView to find the addesses of objects whose values you seek. You then have to know the structure of the information stored there as represented in hex. Only by applying CodeView commands to 32-bit pointers in hex can you inspect the values. So, as you can see, there are substantial limitations to how much debugging can occur at the source level.

Discussion

One question that inevitably comes up with a system such as Objective-C is just how well it stacks up against a more traditional object-oriented system such as Smalltalk. Certainly, if you are using it on a PC-or AT-type machine without the benefit of an interpreter such as Vici, there are some decided limitations to your ability to gain knowledge of the system through exploring it interactively.

In a Smalltalk system, besides the convenience of the browsers and other window-oriented facilities, you can use various built-in methods to help you explore the system and to provide you with information interactively that you need for writing your program. With Objective-C on a PC or AT, you have to rely on written documentation. Fortunately, this documentation is extremely well written, and the difference is essentially one of convenience. As with any programming language, it is possible to write various utilities, such as cross-reference programs and others, that can assist you.

Another issue is the size of the foundation class library that comes with the system. Here, Objective-C lies about midway between C++ and more fully developed environments such as Smalltalk and Actor. One thing that compensates somewhat for this is the ability to use existing C code and libraries to build up a custom class library relatively quickly. This possibility also exists in some cases for other languages too, however.

Keep in mind, too, that the runtime image of Objective-C may not always be compatible with all C programming systems and libraries available for Microsoft C. Unfortunately, this is the case with the Microsoft Windows programming environment. There is apparently a conflict between its run-time requirements and those of Objective-C that makes it impossible to use the two together.

A major difference between Objective-C and other object-oriented languages such as Smalltalk and Actor is the absence of the block construct. In these languages, a block of code is implemented as an unnamed object that allows evaluation to be deferred. Block objects are extremely powerful constructs that allow an additional degree of modularity in defining methods. Research is being conducted on adding the block construct to Objective-C, but it has not as yet been announced as a feature for a future release.

The field of object-oriented programming is besieged with metaphors, and these metaphors can be a mixed blessing. There is no doubt that they perform a useful function in introducing the basic notions of this programming paradigm. But they also can be the source of misconceptions. So, it is important at some point to give them a hard shake to see what fruit they really bear.

In what sense is it really appropriate to compare classes with factories? It is only the products of a class, its instances, that really do anything.

```
#line 2 "demo.c"
typedef struct _PRIVATE * id; id _msg(), _msgSuper();
#line 1 "demo.m"
#line 5 "demo.c"

struct _PRIVATE
 {
  struct _SHARED *isa;
  int xLoc;int yLoc;
 };
  extern id DemoPoint, Object;
  struct _SHARED
 {
  struct _SHARED *isa, *clsSuper;char *clsName;
  char *clsTypes;short clsSizInstance;short clsSizDict;
  struct _SLT *clsDispTable;
 };
 extern struct _SHARED   _DemoPoint, __DemoPoint;
 extern char *Practice[];

#line 1 "demo.m"
#line 3 "demo.m"

/* create=Practice[0] */
 static id _1_DemoPoint(self,_cmd)id self;char *_cmd;
{
 return __msg(_msgSuper(__DemoPoint.clsSuper,Practice[1]
 /*new*/),Practice[2] /*initialize*/);
}
#line 5 "demo.m"
/* initialize=Practice[2] */
 static id _2_DemoPoint(self,_cmd)id self;char *_cmd;
{
 self->xLoc = 100;self->yLoc = 100;return self;
}
#line 7 "demo.m"

/* print=Practice[3] */
static id _3_DemoPoint(self,_cmd)id self;
char *_cmd;
{
 printf("This point's coordinates are (%d@%d ) \n",
 sel  >xLoc,self->yLoc);
}
#line 16 "demo.c"
extern struct _SHARED   _Object, __Object;
struct __SLT
 {
  char **__cmd;id (*__imp)();
 };
  static struct __SLT  _clsDispatchTbl[1]=
 {
  &Practice[0], (id (*)())_1_DemoPoint, /* create */
 };
  static struct __SLT  _nstDispatchTbl[2]=
 {
  &Practice[2], (id (*)())_2_DemoPoint, /* initialize */
  &Practice[3], (id (*)())_3_DemoPoint, /* print */
 };
  static char _bufClsName[]="_DemoPoint";
  struct _SHARED __DemoPoint=
 {
  &__Object,
  &__Object,&_bufClsName[0], 0,sizeof(struct _SHARED), 1,
  (struct _SLT *)_clsDispatchTbl
 };
  struct _SHARED _DemoPoint=
 {
  &__DemoPoint, &_Object,&_bufClsName[1],
  "#ii",sizeof(struct _PRIVATE), 2,
  (struct _SLT *)_nstDispatchTbl
 };
#line 8 "demo.m"
```

Example 3: *C code output of Objective-C compiler*

Like a factory, the class is designed to produce a specific product. But does a factory ever resemble its products? Is a class designed purposely to produce massive numbers of instances efficiently? This tells us right away that a class is more like a mold or template used in a factory rather than the factory itself. However, they are most unique in that they are molds that can also be used to produce other molds as well. In this sense they are more flexible than either factories or molds.

What about the Software-IC metaphor? The idea here is that, just as functional hardware modules can be reused in many different boards, so can classes in a software design. But in just what sense are classes reusable? Aren't library functions also reused in many different applications that their original authors never foresaw? It is important to recognize that classes are reusable in ways that neither library functions or hardware ICs are. You cannot generally use off-the-shelf chips to construct new chips. But you can clearly build new library functions and classes out of existing ones. The real difference between classes and library functions is that classes are much larger and more complex units and are used in very different ways. A class usually has many library functions and data objects contained in it that will belong together in any application in which they are used. Library functions and hardware ICs both fall far short of this.

One of the best features of the Objective-C language is its syntax. It is far more readable than C++ or C. Many people have criticised C for its poor readability. Here, I think, is one area in which Objective-C represents a clear-cut advance over its mother tongue.

Another is that you get many of the advantages of C but on a much higher level. With so many data structures and functions provided as standard in the language, Objective-C in principle allows some fairly large, high-level applications to be written without the danger that much of them will depend on machine-specific library routines or custom code that becomes obscure with the passsage of time.

Finally, Objective-C implements dynamic binding in a C compiler environment, which means that objects are created at run time rather than at compile time. This is clearly one of the features that all AI languages have in common. It is difficult to imagine writing programs that can learn or respond to new situations when everything has been defined statically at the time the program was compiled. This, aside from the more readable syntax, is what really differentiates Objective-C from C++.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Vendor

Objective-C
The Stepstone Corp.
27 Glen Rd.
Sandy Hook, CT 06482
203-426-1875
Reader Service **No. 9**

C FUNCTIONS

Listing One *(Text begins on page 30.)*

```
/*
 * bldfuncs.c: Construct a table of the functions defined in source files.
 * Copyright (c) 1988 Marvin Hymowech
 *
 * Usage:   bldfuncs file1.c file2.c ...
 *          (wildcards are allowed also, e.g. *.c )
 *          The output table is named funcs.txt.
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define LINT_ARGS

#define TRUE      1
#define FALSE     0
#define OK        0
#define ERROR     1

/* return values for filter functions below.
 * EOF or any character is also possible.
 */
#define BRACES    -2
#define PARENS    -3
#define QUOTES    -4

/* function declarations for type checking */
char *get_fn_name(char *);
int get_names_one_file(char *, FILE *);

main(argc, argv)
int argc;
char **argv;
{
   FILE *fp_out;
   char *current_file;
   int num_files, i;

   if( argc < 2 )
   {
      fprintf( stderr, "wrong number of parameters" );
      exit(1);
   }

   if( (fp_out = fopen("funcs.txt", "w")) == NULL )
   {
      fprintf( stderr, "can't open %s\n", "funcs.txt" );
      exit(1);
   }

   /* build a function list for each file on the command line,
    * and write the list to the file funcs.txt.
    */
   printf( "Creating funcs.txt...\n" );
   num_files = argc - 1;
   for ( i = 1; i <= num_files; i++ )
   {                                    /* tell the user where we're at */
      printf( "%30s: %3d of %3d files\n",
                                 current_file = strlwr(*++argv), i, num_files );
      if( get_names_one_file( current_file, fp_out ) != OK )
      {     /* use strlwr to lower-case the name - cosmetic only */
         fprintf( stderr, "can't process %s", current_file );
         exit(1);
      }
   }

   fclose(fp_out);
   exit(0);
}

/* open the .c file source_file_name, scan it for function definitions,
 * and write a listing to fp_out in the form:
 *          source_file_name:
 *             function-1
 *             function-2 ...
 *             function-n;
 * Return values: OK or ERROR
 */
int
```

```
get_names_one_file(source_file_name, fp_out)
char *source_file_name;
FILE *fp_out;
{
   int line_len, c, got_fn_defn = FALSE;
   #define LINE_LEN 8192
   char line[LINE_LEN], *name_ptr, fn_name[64];
   FILE *fp_source;
                                        /* open the input source file */
   if( (fp_source = fopen(source_file_name, "r")) == NULL )
      return ERROR;
                                        /* write "source file name:" */
   sprintf( line, "\n%s:", source_file_name );
   fprintf( fp_out, line );

   while( TRUE )
   {
      line_len = 0;             /* using the filter_data() char stream */
                                /* collect chars until a semicolon or PARENS */
      while( (c = filter_data(fp_source)) != EOF && c != ';' && c != PARENS )
         line[line_len++] = c;
      if( c == EOF )
         break;
      if( c == ';' )            /* Bypass externals representing data items. */
         continue;              /* Now we know line ended with PARENS. */
      line[ line_len ] = 0;     /* Terminate line. */

         /* At this point, line either contains a function definition
          * or a function type declaration or something else, perhaps
          * an occurrence of parenthese in a data item definition.
          * We only want function definitions.
          */
                        /* Extract the function name from this possible */
                        /* function definition, and save it in fn_name. */
      strcpy( fn_name, get_fn_name(line) );
                        /* Exclude the case of parenthetical expressions    */
                        /* in a data defintion, e.g. within array brackets. */
      if( !fn_name[0] )
         continue;
                                  /* skip white space */
      while( (c = filter_data(fp_source)) != EOF && isspace(c) )
         ;
      if( c == ';' || c == ',' )  /* functions type check declaration */
         continue;                /* so bypass it */
      if( c == EOF )
         break;
                                  /* skip any parameter declarations - */
      while( c != BRACES && c != EOF )    /* eat until braces or EOF */
         c = filter_data(fp_source);

                     /* append this function definition to the output table */
      fprintf( fp_out, "\n\t%s", fn_name );
      got_fn_defn = TRUE;
   }
   fclose(fp_source);
                              /* got_fn_defn FALSE if no functions in file */
                               /* write file terminator */
   fputs( got_fn_defn ? ";\n" : "\n\t;\n", fp_out );
   return OK;
}

/* assuming that the input line ends with a function name,
 * extract and return this name (as a pointer into the input line).
 */
char *
get_fn_name(line)
char *line;
{
   char *name_ptr;
   int len;

   if( !(len = strlen(line)) )
      return line;

   name_ptr = line + len - 1;

   while( isspace(*name_ptr) )       /* skip trailing white space */
      name_ptr--;
   *(name_ptr + 1) = 0;              /* terminate fn name */

                                  /* function names consist entirely of */
                                  /* alphanumeric characters and underscores */
   while( (isalnum(*name_ptr) || *name_ptr == '_') && name_ptr >= line )
      name_ptr--;
                                  /* if this alleged function name begins */
   if( isdigit(*++name_ptr) )     /* with a digit, return an empty string */
      return( name_ptr + strlen(name_ptr) );
   return name_ptr;               /* else return the function name */
}
 /* using the stream returned by filter_parens() as input,
  * return a character stream in which any data initialization
  * expressions between an equals sign and a semicolon
  * have been replaced by a single semicolon.
  * This will filter out anything involving parentheses in a
  * data initialization expression, which might otherwise be
  * mistaken for a function defintion.
  */
 int filter_data(fp_source)
 FILE *fp_source;
 {
     int c;

     if( (c = filter_parens(fp_source)) != '=' )
        return c;
     while( (c = filter_parens(fp_source)) != ';' && c != EOF )
        ;
     return c;
  }

  /* using the stream returned by filter_curly_braces() as input,
   * return a character stream in which all characters
   * between '{' and the matching '}' have been replaced
   * by the single special value PARENS (any nested parentheses within
   * this top level pair will also have been eaten).
   * This will filter out anything within the parentheses delimiting
   * the arguments in a function definition.
   */
  int
  filter_parens(fp_source)
  FILE *fp_source;
  {
     int paren_cnt, c;

     if( (c = filter_curly_braces(fp_source)) != '(' )
        return c;
     paren_cnt = 1;
     while( paren_cnt )
        switch( filter_curly_braces(fp_source) )
        {
           case ')':
              paren_cnt--;
              break;
           case '(':
              paren_cnt++;
              break;
           case EOF:
              return EOF;
        }
     return PARENS;
  }

  /* using the stream returned by filter_ppdir() as input,
   * return a character stream in which all characters
   * between '{' and the matching '}' have been replaced
   * by the single special value BRACES (any nested braces within
   * this top level pair will also have been eaten).
   * This will filter out anything internal to a function.
   */
  int
  filter_curly_braces(fp_source)
  FILE *fp_source;
  {
     int brace_cnt, c;

     if( (c = filter_ppdir(fp_source)) != '{' )
        return c;
     brace_cnt = 1;
     while( brace_cnt )       /* wait for brace count to return to zero */
        switch( filter_ppdir(fp_source) )
        {
           case '}':
              brace_cnt--;    /* subtract right braces */
              break;
           case '{':
              brace_cnt++;    /* add left braces */
              break;
           case EOF:
              return EOF;
        }
     return BRACES;           /* brace count is now zero */
  }

  #define MAXLINE 1024

  /* using the stream returned by filter_quotes() as input,
   * return a character stream in which all preprocessor
   * directives have been eaten.
   */
  int
  filter_ppdir(fp_source)
  FILE *fp_source;
  {
     int c, i;
     char line[MAXLINE + 1];

     while(TRUE)
     {                        /* does this line begin a preprocessor directive? */
        if( (c = filter_quotes(fp_source)) != '#' )


        return c;       /* no, return character */
                        /* yes, store until newline or EOF */
     if( (c = get_ppdir_line( fp_source, line )) == EOF )
        return EOF;
     if( strncmp( line, "define", 6 ) )          /* if not #define directive */
        continue;                                /* eat this line */
     if( line[ strlen(line) - 1 ] != '\\' )      /* if #define line ends */
        continue;                                /* with "\" */
     else
        while(TRUE)                              /* keep eating lines */
        {                                        /* which also end with "\" */
                                          /* store until newline or EOF */
           if( (c = get_ppdir_line( fp_source, line )) == EOF )
              return EOF;
                           /* done with this #define directive if this */
                           /* line is not also a continuation line */
           if( line[ strlen(line) - 1 ] != '\\' )
              break;
        }
  }

  /* Utility routine used by filter_ppdir() -
  /* read the character stream using filter_quotes, storing characters
   * in the parameter "line", until EOF or '\n' is encountered.
   * Return EOF or '\n' accordingly.
   */
  int
  get_ppdir_line(fp_source, line)
  FILE *fp_source;
  char *line;
  {
     int i, c;
                                          /* store until newline or EOF */
     for( i = 0; i < MAXLINE && (c = filter_quotes(fp_source)) != '\n'
                                                  && c != EOF; i++ )
        line[i] = c;
     line[i] = 0;                         /* terminate string */
     if( c == EOF )
        return EOF;
     return '\n';
  }

  /* using the stream returned by filter_cmt() as input,
   * return a character stream in which any quoted character
   * or quoted string has been collapsed to the single special value QUOTES
   * to avoid considering special characters like '(', ')', '(', or ')'
   * which may occur within quotes.
   */
  int
  filter_quotes(fp_source)
  FILE *fp_source;
  {
     int c1, c2;

     if( (c1 = filter_cmt(fp_source)) != '\'' && c1 != '"' )
        return c1;     /* pass char through if not single or double quote */
     while( TRUE )
```

```
        switch( c2 = filter_cmt(fp_source) )
        {
           case '\\':                  /* beginning of an escape sequence */
              filter_cmt(fp_source);   /* so eat next char */
              break;
           case EOF:
              return EOF;
           default:
              if( c2 == c1 )           /* found end of quoted char or string */
                 return QUOTES;
        }
    }

    /* Returns character stream, eating comments. */

    /* Returns EOF if end of file. */
    /* Nested comments are allowed. */
    int
    filter_cmt(fp_source)
    FILE *fp_source;
    {
       /* values for state */
       #define STABLE       0        /* not in process of changing the comment */
                                     /* level: i.e., not in the middle of a    */
                                     /* slash-star or star-slash combination.  */
       #define IN_CMT_FS    1        /* got '/', looking for '*' */
       #define OUT_CMT_STAR 2        /* got '*', looking for '/' */

       int c, state = STABLE, cmt_level = 0;

       while( TRUE )
       {
          c = fgetc(fp_source);

          if( c == EOF )
             return EOF;

          switch(state)
          {
             case STABLE:
                if( c == '*' )
                   state = OUT_CMT_STAR;
                else if( c == '/' )
                   state = IN_CMT_FS;
                break;

             case IN_CMT_FS:
                if( c == '*' )
                {
                   state = STABLE;
                   cmt_level++;            /* descend one comment level  */
                   continue;
                }
                else if( !cmt_level )      /* if '/' not followed by '*' */
                {                          /* and outside any comment    */
                   ungetc( c, fp_source ); /* push back this char        */
                   return '/';             /* and return the '/'         */
                }
                else if( c != '/' )        /* stay in state IN_CMT_FS    */
                   state = STABLE;         /* if next char is '/' as well */
                break;

             case OUT_CMT_STAR:
                if( c == '/' )
                {
                   cmt_level--;          /* ascend one comment level */
                   state = STABLE;
                   continue;
                }
                else if( !cmt_level )      /* if '*' not followed by '/' */
                {                          /* and outside any comment    */
                   ungetc( c, fp_source ); /* push back this char        */
                   return '*';             /* and return the '*'         */
                }
                else if( c != '*' )        /* stay in state IN_CMT_FS    */
                   state = STABLE;         /* if next char is '*' as well */
                break;
          }
          if( state == STABLE && !cmt_level )    /* if outside any comment */
             return c;                           /* return character */
       }
    }
```

End Listing One

Listing Two

```
NAME:    bldfuncs - construct a table of C source file names together with
         a list of the names of all functions defined in these source files.

USAGE:   bldfuncs source_file_1 source_file_2 ...
         (wildcard characters are allowed)

DESCRIPTION:     bldfuncs is used to construct the list file needed by getf.
         The output file is named "funcs.txt", and has the format:

                source_file_1.c:
                        function_1
                        function_2
                        ...
                        function_n;
                source_file_2.c:
                        ...

         Of course this file could be constructed or modified using a
         text editor, but bldfuncs is designed to automate this process.
         bldfuncs is run at relatively infrequent intervals to update
         the list file, whereas getf is run frequently to locate functions.

EXAMPLES:        bldfuncs *.c              This will construct a funcs.txt file
                                           in the current directory which will
                                           contain a list of all .c files in
                                           that directory, together with a list
                                           of the functions defined in these
                                           files.

                 bldfuncs *.c \source1\*.c \source2\*.c
                                           This will construct a funcs.txt file
                                           in the current directory which will
                                           contain the above information for
                                           the current directory, \source1,
                                           and \source2 combined.

                 bldfuncs prog1.c prog2.c prog3.c
                                           This will construct a funcs.txt file
                                           in the current directory which will
                                           contain the above information for
                                           the selected files prog1.c, prog2.c
                                           and prog3.c combined.
```

End Listing Two

Listing Three

```
/*
 * getf.c - locate the source file containing a specified function and
 *          present it in your favorite editor.
 * Copyright (c) 1988 Marvin Hymowech
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define LINT_ARGS

#define TRUE      1
#define FALSE     0
#define LINE_LEN 256

   /* function declarations */
int patn_match(char *, char *);
void edit( char *, char * );
void ask_for_file(char **, char **, unsigned int);

unsigned int num_ctl_patns;       /* number of %s symbols in GETFEDIT var */
char *file_token, *func_token;
char argl[64];                    /* argument for editor command line */
char *argl_ctl;                   /* ctl string for building argl */
char *pgm_name;                   /* name of editor to invoke */

main(argc, argv)
int argc;
char **argv;
{
   char file_line[LINE_LEN], func_line[LINE_LEN];
   char func_name[64], edit_cmd[128];
   char *env_edit_cmd, *ctl_patn;
   unsigned int last_func, len, eof = FALSE;
   static char file_name[] = "funcs.txt"; /* input file name */
   FILE *funcs_file;
   static char delim[] = "\n\t ";   /* white space delimiters */
   #define MAX_CHOICES 256
   unsigned int num_choices = 0;
   char *func_choices[ MAX_CHOICES ], *file_choices[ MAX_CHOICES ];


   if( argc != 2 )
   {
      fprintf( stderr, "getf: wrong number of arguments" );
      exit(1);
   }

      /* the argument is the function name to find */
   strcpy( func_name, *++argv );

   if( (env_edit_cmd = getenv( "GETFEDIT" )) == NULL )
   {
      fprintf( stderr, "getf: missing environment variable GETFEDIT" );
      exit(1);
   }

      /* check if GETFEDIT environment variable has a place for
       * function name as well as file name - note function name
       * is assumed to go in the first %s pattern if there are
       * two %s patterns.
       */
   if( (ctl_patn = strstr( env_edit_cmd, "%s" )) == NULL )
   {
      fprintf( stderr,
               "getf: environment variable GETFEDIT has no %%s pattern" );
      exit(1);
   }

   num_ctl_patns = strstr( ++ctl_patn, "%s" ) == NULL ? 1 : 2;
   strcpy( edit_cmd, env_edit_cmd );
   if( (pgm_name = strtok( edit_cmd, " " )) == NULL )
   {
      fprintf( stderr,
               "getf: environment variable GETFEDIT has incorrect format" );
      exit(1);
   }
         /* point to argument following program name */
   argl_ctl = edit_cmd + strlen(pgm_name) + 1;

   if( (funcs_file = fopen( file_name, "r" )) == NULL )
   {
      fprintf( stderr, "getf: can't open %s\n", *argv );
      exit(1);
   }

   while( !eof )      /* loop thru file names, which end with colon */
   {
      if( fgets( file_line, LINE_LEN, funcs_file ) == NULL )
         break;
                     /* bypass any line consisting of white space */
      if( (file_token = strtok( file_line, delim )) == NULL )
         continue;

      if( file_token[ len = strlen(file_token) - 1 ] != ':' )
      {
         fprintf(stderr, "getf: incorrect file format on %s", file_name);
         exit(1);
      }
      file_token[ len ] = 0;  /* kill trailing colon */
      last_func = FALSE;   /* set up to detect last func this file */

      while( !eof && !last_func )   /* loop thru func names this file */
      {                             /* last such ends with semicolon */
         if( fgets( func_line, LINE_LEN, funcs_file ) == NULL )
         {
            eof = TRUE;
            break;
         }
                     /* bypass any line consisting of white space */
         if( (func_token = strtok( func_line, delim )) == NULL )
            continue;

         if( func_token[ len = strlen(func_token) - 1 ] == ';' )
         {
            last_func = TRUE;          /* break loop after this one */
```

```
            func_token[ len ] = 0;      /* kill trailing semi-colon */
         }

         if( patn_match( func_name, func_token ) )
         {
            func_choices[num_choices] = strdup( func_token );
            file_choices[num_choices++] = strdup( file_token );
         }
      }
   }

   switch( num_choices )
   {
      case 0:
         fprintf( stderr, "getf: no match for %s in %s",
                                                  func_name, file_name );
         exit(1);
      case 1:
         edit( func_choices[0], file_choices[0] );
      default:
         ask_for_file(func_choices, file_choices, num_choices);
   }
}
/* return TRUE if string s matches pattern patn,
 * or FALSE if it does not.  Allowable wildcards
 * in patn are: ? for any one character, % for
 * any string of characters up to the next underscore
 * or end of string, and * for any string of characters
 * up to end of string.
 */
int
patn_match(patn, s)
char *patn;    /* pattern */
char *s;       /* string */
{
   for( ; *patn; patn++ )
   {
      if( !*s )                  /* if out of s chars, no match */
         return                   /* unless patn ends with * or % */
            ((*patn == '*' || *patn == '%') && !*(patn + 1) );

      switch( *patn )
      {
         case '%':
            while( *s != '_' && *s )
               s++;
            break;
         case '*':
            while( *s++ )
               ;
            break;
         case '?':
            s++;
            break;
         default:
            if( *s != *patn )
               return FALSE;
            s++;
      }
   }

      return *s == 0;
   }

   void
   ask_for_file(func_choices, file_choices, num_choices)
   char *func_choices[], *file_choices[];
   unsigned int num_choices;
   {
      int i;
      char line[LINE_LEN];

      while( TRUE )
      {
         printf( "Which one? (CR to exit)\n" );
         for( i = 1; i <= num_choices; i++ )
            printf( "\t%3d: %20.20s in %-30.30s\n", i, func_choices[i-1],
                                                     file_choices[i-1] );
         printf( "\nEnter number:" );
         fgets( line, LINE_LEN, stdin );

         if( line[0] == '\n' )
            break;
         if( (i = atoi(line)) < 1 || i > num_choices )
            printf( "\nInvalid choice!\007\n" );
         else
            edit( func_choices[i-1], file_choices[i-1] );
      }
   }

   void
   edit(func, file)
   char *func, *file;
   {
            /* execlp will overlay this program with the editor */
      if( num_ctl_patns == 1 )
         sprintf( arg1, arg1_ctl, file );
      else
         sprintf( arg1, arg1_ctl, func, file );
      execlp( pgm_name, pgm_name, arg1, NULL);
                     /* if we're still here, print error msg */
      fprintf( stderr, "getf: exec failed" );
      exit(1);
   }
```

End Listing Three

Listing Four *(Listing continued, text begins on page 30.)*

```
NAME:    getf - locate the source file containing a C function
                and present it in a specified editor.

USAGE:   getf [list_file_name] function name

DESCRIPTION:     getf looks for the file funcs.txt, which is
         presumed to be a listing of C source file names together
         with a list of the C functions defined in these source
         files; the expected format is

                source_file_1.c:
                        function_1
                        function_2
                        ...
                        function_n;
                source_file_2.c:
                        ...

         (This list file can be conveniently built using the companion
         program bldfuncs.c.)

                The editor to be used is specified in the DOS environment
         variable GETFEDIT, which is presumed to have the form:

                editor_name  control_string

         where control_string is an sprintf control string which may have
         either one or two occurrences of the pattern "%s": if one such
         pattern exist, it is replaced by the appropriate source file
         name in which the requested function resides; if two such
         patterns exist, the first "%s" is replaced by the function name
         and the second by the source file name.  This latter format allows
         the use of an editor-specific search command to position the
         source file to the requested function automatically.  After
         replacing the "%s" patterns as above, the GETFEDIT string is
         executed via a DOS exec call, thus overlaying the current program
         getf.

         The wildcards ?, * and % are allowed in the function name
         argument for getf, and are interpreted as: ? matches any single
         character, * matches the remainder of any string, and % matches
         all characters up to the next underscore or until the end of
         string.

EXAMPLES:  Assume the editor is BRIEF; possible GETFEDIT strings might be:

                b %s                    this will simply bring up the
                                        desired source file in BRIEF
                                        without any attempt at positioning
                                        the cursor to the requested function.

                b -m"search_fwd %s" %s  this will bring up the desired
                                        source file and position the cursor
                                        to the first occurrence of the
                                        requested function.

                b -m"funcsrch %s" %s    (see funcsrch.doc for an explanation
                                        of this customized BRIEF macro.)
                                        this will bring up the desired
                                        source file, seek to the end of
                                        the file, then seek backwards for
                                        the LAST occurrence of the requested
                                        function.  In practice, this is the
                                        approach most likely to position the
                                        cursor to the actual definition of
                                        the function.

         To illustrate wildcards:

                getf func*              matches func, func1, func_2;

                getf func%              mathces func, func1, but not func_2.

                getf func?_%_*          matches func1_new_, funca_old_stuff,
                                        but not func1_stuff.

NOTE:    Remember that to set GETFEDIT in your autoexec.bat file (or in
         any other batch file) you must use TWO percent signs instead of
         one:
                set GETFEDIT=b -m"funcsrch %%s" %%s

         This is necessary to avoid interpretation of the percent signs
         by command.com.
```

End Listings

DOCUMENTING C

Listing One *(Listing continued, text begins on page 40.)*

```
/* print a visual tree representation of a 'C' program */
/* cp.c */

/*************************************************************

  cprinter - print a visual tree representation
             of a 'C' program

  copyright 1987, Stewart A. Nutter

  Please do not distribute this for profit. For
  individual use only.

  written by: Stewart A. Nutter

 *************************************************************/

#define MAINMODULE 1
#include "cpheader.h"

main( argc, argv )

char **argv;
int argc;

{
char szName[20];              /* input file name */
int iRet;
int l, i, j, k;               /* index variables */
int sflag;
int pcnt;
int tmp;
int pmax;                     /* max number of xref columns */
int index;
FILE *stream;
struct Pages *p;
long int total;               /* total number of bytes */
long int ltotal;              /* total number of lines */

pFlist = Flist;
pMlist = Mlist;
pMnames = Mnames;
```

```
for ( i=0; i<50; i++ )      /* clear out the recursion list */
   rlist[i] = NULL;

printf( "\ncp - ver. 1.3, (C) 1987, 1988  Stewart A. Nutter" );
printf( "\n     written by Stewart A. Nutter\n" );

/* no arguments - print instructions */

if ( argc < 2 )
   {
   printf( "\ncp listfile [ outfile ] [ /l:xx /w:yy /t:main /s:z ]\n" );
   printf( "      outfile           = \"prn\" \n" );
   printf( "      l: page length    = 66       [0, 50-255]\n" );
   printf( "      w: page width     = 80       [80-255]\n" );
   printf( "      m: left margin    = 8        [0-30]\n" );
   printf( "      r: right margin   = 8        [0-30]\n" );
   printf( "      t: target function = \"main\"\n" );
   printf( "      s: statistics only = 0       [0-all, 1-stats only]\n" );
   printf( "\n" );
   printf( "Notes: 1. Maximum recursive function displacement of $0.\n" );
   printf( "       2. Maximum number of functions calls is %d.\n", MAXFNCTS );
   printf( "       3. Maximum number of modules is %d.\n", MAXMODULES );
   exit( 0 );
   }


if ( ( stream = fopen( argv[1], "r" ) ) == NULL )
   {
   fprintf( stderr, "\n%s", strerror( errno ) );
   exit( 1 );
   }

/* an output file name was given? */

index = 2;

if ( argc > 2 && argv[index][0] != '/' )
   {
   output = fopen( argv[2], "wt" );
   index++;
   }
else
   output = fopen( "prn","wt" );  /* prn device by default */

for ( i=index; i<argc; i++ )
   {
   if ( argv[i][0] == '/' && strlen( argv[i] ) > 3 && argv[i][2] == ':' )

        {
        switch ( argv[i][1] )
           {
           case 'l' :                /* change the page length */
              tmp = atoi( &argv[i][3] );
              if ( ( tmp > 50 && tmp < 256 ) || tmp == 0 )
                pl = tmp;
              break;

           case 'm' :                /* change the left margin */
              tmp = atoi( &argv[i][3] );
              if ( tmp >= 0 && tmp <= 30 )
                lm = tmp;
              break;

           case 'r' :                /* change the rignt margin */
              tmp = atoi( &argv[i][3] );
              if ( tmp >= 0 && tmp <= 30 )
                rm = tmp;
              break;

           case 's' :                /* set the stats only? */
              stats = atoi( &argv[i][3] );
              break;

           case 't' :                /* change the target function */
              strcpy( target, &argv[i][3] );
              break;

           case 'w' :                /* change the width */
              tmp = atoi( &argv[i][3] );
              if ( tmp > 79 && tmp < 256 )
                pw = tmp;
              break;

           }
        }
     else
        {
        printf( "\nUnknown argument: %s", argv[i] );
        exit( 1 );
        }
     }

 if ( output == NULL )
    {
    fprintf( stderr, "\n%s", strerror( errno ) );
    exit( 1 );
    }

 width = pw - lm - rm;
 if ( width < 40 )
    {
    fprintf( stderr, "\nThe page width is too narrow." );
    exit( 1 );
    }

 printf( "\n" );


 /* read the input file for file names */

while ( !feof( stream ) )
   {
   szName[0] = '\0';
   fgets( szName, 19, stream );
   if ( ( l = strlen( szName ) ) > 1 )
      {
      if ( szName[l - 1] == '\n' )
         szName[l - 1] = '\0';    /* remove newline char */
      xref( szName );
      }
   }

/* pointer list for sort */

for ( i=0, pMlist=Mlist; i<Mqty; i++ )
   {
   pm[i] = pMlist++;
   }

printf( "\n\nSorting the function list...\n" );

sflag = 1;
while ( sflag )                  /* sort the function names */
   {
   sflag = 0;
   for ( i=0; i<Mqty-1; i++ )
      {
      if ( strcmp( pm[i]->function, pm[i+1]->function )>0 )
         {
         sflag = 1;
         pMlist = pm[i];
         pm[i] = pm[i+1];
         pm[i+1] = pMlist;
         }
      }
   }
i = find_mod( target );  /* must start with the target function */

if ( i >= 0 )                    /* 'main' must exist */
   {
   depth = 0;

   printf( "Checking for usage...\n" );

/* check how many times each function is used */

   getstats( );
   depth = 0;
   bfr[0] = 0;

   printf( "Starting the printout...\n" );

   line = 0;
   if ( stats == 0 )
      {
      pm[i]->used = 1;   /* set so main shows up in the list */
      doprint( i );  /* print the non-library functions */

      for ( i=0; i<Mqty; i++ )  /* print defined functions now */
         {
         fprintf( output, "\n" );
         line++;
         if ( pm[i]->used > 1 ) /* must be used more than once */
            {
            doprint( i );          /* print the tree structure */
            }
         }
      }

/* print statistics on the modules */

   line = 9999;                  /* force a new page */
   pMnames = Mnames;
   pagebreak( );
   leftmargin( output );
   fprintf( output, "Module statistics :\n" );
   line++;
   total = 0L;
   ltotal = 0L;
   for ( i=0; i<Mcnt; i++ )  /* print module names & sizes */
      {
      pagebreak( );
      leftmargin( output );
      fprintf( output,
              "%-12s - %5u lines, %6ld bytes\n",
              pMnames->name, pMnames->length,
              pMnames->size );
      total += pMnames->size;
      ltotal += pMnames->length;
      line++;
      pMnames++;
      }
   fputc( '\n', output );
   leftmargin( output );
   fprintf( output,
          "Total source size = %ld bytes in %ld lines for %d
                                                  modules\n",
           total, ltotal, Mcnt );

/* print the used function page index */
```

```
   line = 9999;                    /* force a new page */
   pagebreak( );
   leftmargin( output );
   fprintf( output, "Function index :\n" );
   line++;
   for ( i=0; i<Mqty; i++ )     /* print used function names */
      {
      pMlist = pm[i];
      if ( pMlist->used > 0 )
         {
         pagebreak( );
         leftmargin( output );
         fprintf( output,
                "%-25s - %-12s - used =%d \n",
                pMlist->function, ( pMlist->name )->name,
                pMlist->used );
         line++;
         }
      }

/* print the function page cross reference */
   if ( stats == 0 && pl > 0 )      /* print everything */
      {
      pmax = ( int )( width - 27 )/5;
      line = 9999;                    /* force a new page */
      pagebreak( );
      leftmargin( output );
      fprintf( output, "Function cross reference :\n" );
      line++;
      for ( i=0; i<Mqty; i++ )  /* print used function names */
         {
         pMlist = pm[i];
         if ( pMlist->used > 0 )
            {
            pagebreak( );
            leftmargin( output );
            fprintf( output, "%-25s- ", pMlist->function );
            p = pMlist->pg;
            if ( p != NULL )
               {
               pcnt = 0;
               while ( p->next != NULL )
                  {
                  fprintf( output, "%4d,", p->pg );
                  p = p->next;
                  pcnt++;
                  if ( pcnt >= pmax )
                     {
                     fputc( '\n', output );
                     leftmargin( output );
                     fprintf( output, "%27s", " " );
                     line++;
                     pcnt = 0;
                     }
                  }
               fprintf( output, "%4d\n", p->pg );
               line++;
               }
            else
               fprintf( output, "\n" );
            }
         }
      }

/* print statistics on all unused modules */

   line = 9999;                    /* force a new page */
   pagebreak( );
   leftmargin( output );
   fprintf( output, "Un-used function list :\n" );
   line++;
   pcnt = 0;
   for ( i=0; i<Mqty; i++ )  /* print unused function names */
      {
      pMlist = pm[i];
      if ( pMlist->used == 0 )
         {
         pagebreak( );
         pcnt++;
         leftmargin( output );
         fprintf( output,
                "%-25s - %-12s \n",
                pMlist->function, ( pMlist->name )->name );
         line++;
         }
      }
   if ( pcnt == 0 )
      {
      leftmargin( output );
      fprintf( output,
             "No un-used functions in the list.\n" );
      }

/* print module comments */

   line = 9999;                    /* force a new page */
   pMnames = Mnames;
   pagebreak( );
   leftmargin( output );
   fprintf( output, "Module comments :\n" );
   line++;
   for ( i=0; i<Mcnt; i++ )  /* print module names & comments */
      {
      pagebreak( );
      leftmargin( output );
      fprintf( output,
             "%12s -%s\n",
             pMnames->name, pMnames->cmt );
      line++;
      pMnames++;
      }
   fprintf( output, "%c", 0x0c );          /* ending formfeed */
   }
}
/* process the file for function names */

xref( fname )

char *fname;

{
int done;               /* loop termination flag */
int brace_cnt;          /* count of the open braces */
int open_paren;         /* open paranthisis count */
int ret;                /* return value */
int indx;               /* for/next index */
int dflg;               /* function definition flag */
static int wflg = 0;
char c;                 /* character read from disk file */
char buffer[50];        /* temporary buffer */
char bufr[256];         /* temporary buffer */
register char *p;       /* fast character pointer */
FILE *stream;           /* module file pointer */
struct Mod_list *cptr;  /* pointer to the module list structure */
static char back[] =
   {8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8};

printf( "%cProcessing file: %-12s   ", 0x0d, fname );

if ( ( stream = fopen( fname, "r" ) ) == NULL )
   return( -1 );

pp = pc;
if ( ( pMnames->name = strdup( fname ) ) == NULL )
   {
   fprintf( stderr, "Ran out of memory.\n" );
   exit( 1 );
   }
pMnames->length = 0;
pMnames->size = 0L;

buffer[0] = 0;
p = buffer;

open_paren = 0;
brace_cnt = 0;
firstcmt = 0;
filecmt[0] = 0;
done = 0;
ret = 0;
while ( !ret )
   {
   c = getnext( stream );
   switch ( c )
      {
      case '{'  :
         brace_cnt++;   /* increment the open brace count */
         break;
      case '}'  :
         brace_cnt--;   /* decrement the open brace count */
         break;
      case '('  :
         if ( open_paren == 0 ) /* first open paren only */
            {
            open_paren = 1;
            }
         wflg = 1;
         break;
      case ' ' :              /* skip tabs and spaces */
      case '\t' :
         do
            {
            c = getnext( stream );
            }
         while ( c == '\t' || c == ' ' );
         if ( c != '(' )
            wflg = 1;
         pushc( c );
         break;
      case 0x1a :             /* end of the file indicator */
         ret = 1;
         wflg = 1;
      default :

/* the character must be a variable character */

         if ( strcheck( c ) )
            {
            *p++ = c;
```

```
            *p = 0;
            }
         else
            wflg = 1;
         break;
      }
   if ( wflg )
      {
      if ( buffer[0] && ( buffer[0] < '0' || buffer[0] > '9' ) )
         {
         done = 1;
         }
      else
         {
         p = buffer;
         buffer[0] = 0;
         open_paren = 0;
         }
      wflg = 0;
      }

/* if done != 0 there is a token */

   if ( done )
      {
      done = 0;
      *p = 0;

      if ( open_paren )                /* functions start with an
                                                  open paren */

         {
         open_paren = 0;
         if ( brace_cnt == 0 )         /* and no braces */
            {
            dflg = 0;
            for ( indx=0; indx<256 && dflg==0; indx++ )
               {
               c = getnext( stream );
               if ( c == ';' )
                  dflg = 1;
               else if ( c == '\n' )
                  dflg = 2;
               bufr[indx] = c;
               }
            if ( dflg == 0 )
               {
               fprintf( stderr, "\nSyntax error: " );
               fprintf( stderr, "Module description.\n" );
               bufr[indx] = 0;
               fprintf( stderr, "\n%s\n", bufr );
               exit( 1 );
               }

/* put the characters back to be read */

            while ( indx )
               {
               pushc( bufr[indx-1] );
               indx--;
               }

/* this is a function definition */

            if ( dflg == 2 )
               {
               printf( "%-40s%s", buffer, back );
               pMlist->name = pMnames;
               pMlist->qty = 0;
               pMlist->ptr = pFlist;

               /* allocate memory for name */

               if ( ( pMlist->function = strdup( buffer ) )
                    == NULL )
                  {
                  fprintf( stderr, "\nRan out of memory." );
                  exit( 1 );
                  }
               pMlist->used = 0;
               pMlist->pg = NULL;
               cptr = pMlist;
               pMlist++;
               Mqty++;
               if ( Mqty > MAXMODULES )
                  {
                  fprintf( stderr,
                           "Too many new functions\n" );
                  exit( 1 );
                  }
               }
            }
         else
            {
            cptr->qty += addlist( cptr->ptr,
                                  buffer, cptr->qty );

            }
         }
      p = buffer;
      *p = 0;
      }
   }
fclose( stream );
pMnames->cmt = strdup( filecmt );
pMnames++;   /* point to the next function data structure */
Mcnt++;              /* count of the different functions */
return( ret );
}
```

End Listing One

Listing Two

```
cp.obj : cp.c cpheader.h
   cl -Ox -AL cp.c -c

cpfuncts.obj : cpfuncts.c cpheader.h
   cl -Ox -AL cpfuncts.c -c

cp.exe : cpfuncts.obj cp.obj
   link cp+cpfuncts/st:4096;
```

End Listing Two

Listing Three

```
/* function module for the program 'cp'
   the cp program must be compiled with the large model */

/*********************************************************

 cpfuncts.c - function module for the program 'cp'

 copyright 1987, Stewart A. Nutter

 written by: Stewart A. Nutter

 *********************************************************/


#define MAINMODULE 0
#include "cpheader.h"
/* getnext - get the next character from the stream */

getnext( stream )

FILE *stream;

{
register char c;
static int qflag=0, cflag=0, eflag=0;
static int dflag=0, aflag=0, ncnt=0;
static int fp;
int b, done;

done = 1;
do
   {
   if ( ( qflag | cflag | eflag | dflag | aflag ) == 0 )
      done = 1;
   c = getchars( stream );

/* process escape sequence characters */

   if ( eflag && c != 0x1a )
      {
      if ( c >= '0' && c <= '7' && ncnt < 3 )
         ncnt++;
      else
         {

/* had less than the 3 octal digits */

         if ( ncnt < 3 && ncnt > 0 )
            pushc( c );
         ncnt = 0;
         eflag = 0;
         }
      }
   else if ( cflag && c != 0x1a )  /* skipping a comment */
      {
      if ( firstcmt == 1 )
         {
         if ( c != '\n' && strlen( filecmt ) < ( width - 14 ) )
            {

            filecmt[fp] = c;
            fp++;
            filecmt[fp] = 0;
            }
         else
            {
            do       /* remove extraneous spaces and tabs */
               {
               b = getchars( stream );
               }
            while ( b == ' ' || b == '\t' );
            pushc( b );
            filecmt[ fp++ ] = ' ';
```

```
            filecmt[ fp ] = '\0';
            }
         }
      if ( c == '*' )
         {
         b = getchars( stream );
         if ( b == '/' )
            {
            firstcmt = 2;      /* done with the comment */
            if ( fp > 0 )
               filecmt[fp - 1] = 0; /* terminate the line */
            cflag = 0;
            }
         else
            pushc( b );
         }
      }
   else if ( qflag && c != 0x1a )  /* skipping a string */
      {
      if ( c == 0x27 )
         eflag = 1;
      else if ( c == '\"' )
         {
         pushc( 0x1b );
         qflag = 0;
         }
      }
   else if ( dflag && c != 0x1a )  /* defines/includes etc. */
      {
   if ( c == '\n' )
      dflag = 0;
   }
else if ( aflag && c != 0x1a )  /* skip a character */
   {
   if ( c == 0x27 )
      {
      aflag = 0;
      pushc( 0x1b );
      }
   else if ( c == '\\' )
      eflag = 1;
   }
else
   {
   switch ( c )
      {
      case '\"' :
         qflag = 1;
         break;
      case 0x27 :
         aflag = 1;
         break;
      case '#' :
         dflag = 1;
         break;
      case '/' :
         b = getchars( stream );
         if ( b == '*' )
            {

            /* this is the first comment of the file */

            if ( firstcmt == 0 )
               {
               firstcmt = 1;
               filecmt[0] = 0;
               fp = 0;
               }
            cflag = 1;
            }
         else
                  {
                  pushc( b );
                  }
               break;
            }
         }
      if ( aflag || dflag || qflag || eflag || cflag )
         done=0;
      }
   while ( !done && c != 0x1a );
   if ( c == 0x1a )
      {
      ncnt = 0;
      aflag = 0;
      dflag = 0;
      qflag = 0;
      eflag = 0;
      cflag = 0;
      }
   return( c );
   }


   /* getchars - read inputs from the file */

   getchars( stream )

   FILE *stream;

   {
   register char c;

   if ( pp != pc )
      c = *--pp;
   else
      {
      c = fgetc( stream );
      if ( c == EOF )
         c = 0x1a;
      if ( c == 0x0a )
         {
         pMnames->length++;
         pMnames->size++;      /* count the unseen <cr> */
         }
      pMnames->size++;
      }
   return( c );
   }
   /* pushc - save the char. in a last in first out stack */

   pushc( c )

   char c;

   {
   if ( ( pp - pc ) < 1000 )
      *pp++ = c;
   else
      {
      fprintf( stderr, "\nProgram syntax error:" );
      fprintf( stderr, " Too many pushed characters.\n" );
      exit( 1 );
      }
   }


   /* addlist - add the name to the list if different */
   /*           and if not one of the 'c' key words */

   #define KEYS 5

   addlist( p, buf, cnt )

   struct Func_list *p;
   char *buf;
   int cnt;

   {
   int i, ret;
   static char *keywords[KEYS] =
      {
      "while",
      "if",
      "for",
      "switch",
      "return",
      };

   for ( i=0; i<KEYS && strcmp( buf, keywords[i] )!=0; i++ )
      ;

   if ( i < KEYS )
      return( 0 );
   for ( i=0; i<cnt && strcmp( buf, p->name ); i++ )
      p++;

   if ( i == cnt )
      {
      ret = 1;
      if ( ( pFlist->name = strdup( buf ) ) == NULL )
         {
         fprintf( stderr, "Ran out of memory.\n" );
         exit( 1 );
         }
      pFlist->used = 1;
      pFlist++;            /* point to the next empty cell */
      Fqty++;
      if ( Fqty > MAXFNCTS )
         {
         fprintf( stderr, "Too many functions.\n" );
         exit( 1 );
         }
      }
   else
      {
      ret = 0;
      p->used++;
      }
   return( ret );
   }


   /* find_mod - return the index of the linked list for
                 the indicated function. A -1 means that
                 the function name was not found in the list */

   find_mod( buf )

   char *buf;

   {
```

```
int lo, hi, mid;
int d;

lo = 0;
hi = Mqty - 1;
mid = ( hi + lo )/2;

while ( 1 )
   {
   d = strcmp( buf, pm[mid]->function );
   if ( d == 0 )
      break;

   if ( lo >= hi )
      {
      mid = -1;
      break;
      }

   if ( d < 0 )
      {
      hi = mid - 1;
      }
   else
      {
      lo = mid + 1;
      }
   mid = ( hi + lo )/2;
   }

return( mid );
}
/* doprint - print the function name and sub - functions */

static char lib[] = {"(library)"};
static char use[] = {"Used="};
static char fct[] = {"Functs="};

doprint( n )

int n;
{
int i, j, k, l, ret;
struct Mod_list *p;
struct Func_list *q;

l = n;
p = pm[l];

/* add function to list for recursion check */
rlist[depth] = p->function;

pagebreak( );

setpage( pm[l] );
ret = page - 1;
pblock( bfr, p->function, ( p->name )->name, fct, p->qty );

depth++;
strcat( bfr, "   |" );

k = p->qty;
for ( j=0, q = p->ptr; j<k; j++, q++ )
   {
   pagebreak( );
   i = find_mod( q->name );

   if ( recur_chk( q->name ) )
      {
      leftmargin( output );
      fprintf( output, "%s\n", bfr );
      if ( i >= 0 )
         setpage( pm[i] );
      pblock( bfr, q->name, "(recursive)", "", 0 );
      line++;
      }
   else
      {
      if ( i >= 0 )
         {
         if ( pm[i]->used == 1 )
            {
/* got a new function */

            leftmargin( output );
            fprintf( output, "%s\n", bfr );
            line++;
            doprint( i );          /* used only once */
            }
         else
            {
/* a previously defined function */

            leftmargin( output );
            fprintf( output, "%s\n", bfr );
            setpage( pm[i] );
            pblock( bfr, q->name, "(defined)",
                    use, q->used );
            line++;
            }
         }
      else
         {
/* a library function */

         leftmargin( output );
         fprintf( output, "%s\n", bfr );
         pblock( bfr, q->name, lib, use, q->used );
         line++;
         }
      }
   }

/* remove the function from the recursion list */

rlist[depth] = NULL;
bfr[strlen( bfr )-4] = 0;
depth--;
return( ret );
}


/* getstats - get the number of times each
              function is used */

getstats( )

{
register int i;
int j;
register struct Func_list *p;

p = Flist;

for ( i=0; i<Fqty; i++ )
    {
    j = find_mod( p->name );  /* see if the name exists */
    if ( j >= 0 )
       pm[j]->used += p->used;
    p++;
    }
}
/* pblock - print a function block */

pblock( pre, fptr, mptr, sptr, cnt )

char *pre, *fptr, *sptr, *mptr;
int cnt;

{
leftmargin( output );
fprintf( output, "%s %s\n", pre, tline );
leftmargin( output );
fprintf( output, "%s-+%-25s|\n", pre, fptr );
leftmargin( output );
fprintf( output, "%s |%-12s %8s%3d |\n",
       pre, mptr, sptr, cnt );
leftmargin( output );
fprintf( output, "%s %s\n", pre, tline );
line += 4;

}

/* pagebreak - check for a page break and if so
               then print the page header */

pagebreak( )

{
   int i;
   static char title[] = { "C PRINTER  - (c) 1987, 1988
                                                rev. 1.2" };
if ( pl == 0 && line == 9999 )
   {
   fprintf( output, "\n\n\n\n\n" );
   line = 0;
   }
else if ( pl != 0 )
   {
   if ( line > ( pl - 11 ) )
      {
      fprintf( output, "%c", 0x0c );
      line = 0;
      }
   if ( line == 0 )
      {
      leftmargin( output );

      fprintf( output, "%s", title );
      for ( i=strlen( title ); i<width-10; i++ )
         fputc( ' ', output );
      fprintf( output, "Page:%4d\n\n", page );

      leftmargin( output );
      for ( i=0; i<width; i++ )
         fputc( '-', output );
      fprintf( output, "\n\n" );
      line = 3;
      page++;
      }
   }
}
```

```
/* recur_chk - check if the function just called
               is one being processed */

recur_chk( buf )

char *buf;

{
register char **p;
int ret;

p = rlist;

while ( *p != NULL && strcmp( *p, buf ) )
   {
   p++;
   }
if ( *p == NULL )
   ret = 0;   /* the function was not in the list */
else
   ret = 1;   /* found it */

return ret;
}


/* setpage - put the current page number
             into the linked page list */

setpage( ptr )

struct Mod_list *ptr;

{
struct Pages *p;

p = ptr->pg;
if ( p == NULL )
   {
   p = ( struct Pages * )malloc( sizeof( struct Pages ) );
   if ( p == NULL )
      {
      fprintf( stderr,
               "Ran out of memory for page # list.\n" );
      exit( 1 );
      }
   ptr->pg = p;
   p->next = NULL;
   p->pg = page - 1;
   }
else
   {
   while ( p->next != NULL )
      p=p->next;
   p->next = ( struct Pages * )malloc( sizeof( struct Pages ) );
   if ( p->next == NULL )
      {
      fprintf( stderr,
               "Ran out of memory for page # list.\n" );
      exit( 1 );
      }
   p = p->next;
   p->next = NULL;
   p->pg = page - 1;
   }
}


/* strcheck - check if the character is
              in the variable name set */

strcheck( c )

char c;

{
if ( ( c >= 'A' && c <= 'Z' ) || ( c >= 'a' && c <= 'z' ) ||
     c == '_' || ( c >= '0' && c <= '9' ) )
   return( 1 );
else
   return( 0 );
}

stop( )
{
printf( "hello" );
}
/* print the left margin for the printout */

leftmargin( output )

   FILE *output;                    /* the output device pointer
                                                              */

{
   register int i;

for ( i=0; i<lm; i++ )
   fputc( ' ', output );
}
```

End Listing Three

Listing Four

```
#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXFNCTS   4000     /* maximum number of functions */
#define MAXMODULES 500      /* number of different files */

struct Func_list            /* function statistics */
   {
   char *name;              /* function name */
   int used;                /* times used in function */
   };

struct Pages                /* linked list of pages */
   {
   int pg;                  /* page number */
   struct Pages *next;      /* pointer to next page */
   };

struct Module               /* module statistics */
   {
   char *name;              /* pointer to the module name */
   char *cmt;               /* module comment/description */
   unsigned int length;     /* lines in the module */
   long size;               /* bytes in the module */
   };

struct Mod_list             /* module control stucture */
   {
   struct Module *name;     /* pointer to module file name */
   char *function;          /* name of function in file */
   struct Func_list *ptr;   /* pointer to the first function */
   int qty;                 /* different functions called */
   struct Pages *pg;        /* point to the page list */
   int used;                /* times the function is used */
   };

#if MAINMODULE != 0

struct Module Mnames[MAXMODULES], *pMnames;
struct Mod_list Mlist[MAXMODULES], *pMlist, *pm[MAXMODULES];
struct Func_list Flist[MAXFNCTS], *pFlist;
char *rlist[50];           /* recursion function list */
int Mqty = 0, Fqty = 0, Mcnt = 0;
int page=1, line=0, depth=0;
int fline;
int firstcmt;
char bfr[300];
char tline[] = {"+-------------------------+"};
char dbl[] = {"  "};
char vbar[] = {"| "};
char hbar[] = {"+-"};
FILE *output;
char pc[1000] = {0,0,0,0,0,0,0,0,0,0};
char *pp;
char filecmt[300];
int pw = 80;               /* page width */
int pl = 66;               /* page length */
int lm = 8;                /* left margin */
int rm = 8;                /* right margin */
int width;                 /* the printable page width */
char target[40] = "main";  /* target function */
int stats = 0;

#else

extern struct Module Mnames[MAXMODULES], *pMnames;
extern struct Mod_list Mlist[MAXMODULES], *pMlist;
extern struct Mod_list *pm[MAXMODULES];
extern struct Func_list Flist[MAXFNCTS], *pFlist;
extern char *rlist[];       /* recursion function list */
extern int Mqty, Fqty, Mcnt;
extern int page, line, depth;
extern int fline;
extern int firstcmt;
extern char bfr[];
extern char tline[];
extern char dbl[];
extern char vbar[];
extern char hbar[];
extern FILE *output;
extern char pc[];
extern char *pp;
extern char filecmt[];
extern int pw;                      /* page width */
extern int pl;                      /* page length */
extern int lm;                      /* left margin */
extern int rm;                      /* right margin */
extern int width;                   /* the printable page width */
extern char target[];               /* target function */
extern int stats;

#endif
```

End Listings

Listing One *(Text begins on page 50.)*

```
/* this routine uses a supplied file (n>64K) of random one byte numbers to generate a set of
encryption and decryption rotors. These rotors are supplied sequentially in a 128K file. The first
64K of the file contains the encryption rotors in series, the second 64K contains the decryption
rotors in series. The order within the file is the 256 elements of the first rotor in sequence
followed by the 256 elements of the second rotor, on through all the rotors in the set.

Since random files are not easily created by users it is necessary to create a close approximation
to random files. This can be done by taking a variety of text and program files, compressing them,
encrypting them with random keys, and concatenating them to form a long binary file.  If English
text was the source material this compressed, binary file should be at least 256K bytes long (making
allowances for the redundancy of English). The resulting 4:1 overlap of compressed English removes
the redundancies and residual order present in the original language file. The length of the binary
file should be entered into the #define NN line since the binary nature of the file prevents EOF
characters. Use command line file direction to read the random file into the program and direct the
output into the rotor file. */

#include <stdio.h>
#define NN  /* length of random file in bytes */

main()
    {
    unsigned char rnd[65536];              /* random number array          */
    unsigned char rotor[2*65536];
    register int k, temp, c;
    register long i, j;

    for(i=0, j=0; i<NN; i++)
        {
        rnd[j++] ^= ((unsigned char)(c=getchar()));
        if (j==65536)
            j = 0;
        }

    for(i=0; i<65536; i+=256)
        for(j=0; j<256; j++)
            rotor[i+j] = j;

    for(i=0; i<65536; i+=256)
        for(j=0; j<256; j++)
            {
            k=rnd[i+j];
            temp = rotor[i+j];
            rotor[i+j] = rotor[i+k];
            rotor[i+k] = temp;
            rotor[65536 + i + rotor[i+j]] = j;
            rotor[65536 + i + rotor[i+k]] = k;
            }

    for(i=0; i<2*65536; i++)
        putchar(rotor[i]);
    }
```

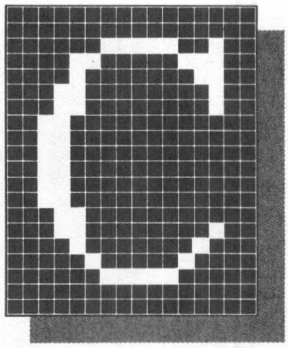
End Listing One

Listing Two

```
/* in the following subroutines the PR numbers used for GR operations are stored in rndout[] and are
accessed by the offset values ro. These values are incremented by 8 for each character processed to
allow for the 8 PR numbers needed per GR operation. */

/*________ character substitution encryption _________________*/
unsigned char sub(ch,ro)
unsigned char ch;       /* character to be encrypted */
int ro;                 /* offset in random number array */
    {
    int i;

    for(i=0;i<4;i++)
ch=rotor[(int)((rndout[i+ro])<<8)+(int)((rndout[i+4+ro]+ch)&255)];
    return(ch);
    }

/*__________ character substitution decryption _____________*/
unsigned char unsub(ch,ro)
unsigned char ch;       /* character to be decrypted */
int ro;                 /* offset in random number array */
    {
    int i;

    for(i=3;i>=0;i--)
ch=(rotor[65536+(int)((rndout[i+ro])<<8)+(int)ch]-rndout[i+4+ro])&255;
    return(ch);
    }
```

End Listings

Off and Running . . .



by Al Stevens

When the folks at *DDJ* called me about doing this column, they offered me the chance to repay a long overdue debt. As a loyal *DDJ* reader since its earliest days, I have received much knowledge, advice, and good code from its pages. This column then is the first installment in what could become a long period of repayment. Please, now, indulge me in a bit of introduction.

I started programming 30 years ago when real pilots flew taildraggers, real beer cans were made of steel, and real computers had drum memory and vacuum-tube logic circuits. This statement probably says more about my age than my qualifications because little of what I learned during those early years applies in today's software-development profession. Programming wasn't considered a profession or even a career then, and no one worried much about whether programming was an art or a science. It was a job. There were no university courses, no structure or discipline, and no magazines. Nobody published or shared source code. *MAD* magazine was six years old, *Playboy* was five, and *DDJ* was not yet a gleam. As my old pal Bill Chaney says, "Times was tough, Buddy." Times are better now, Bill.

My brother Fred introduced me to *DDJ* when its title was much longer and more whimsical. Fred built one of the early Altair 8080 computers in about 1976. He built it to use in developing microprocessor-based systems. Fred and I weren't hobbyists or hackers in the usual sense—our needs were more mundane, and although we had no association with hackers, we didn't lack the hacker's enthusiasm. Our combined consulting activities—his in hardware design and mine in software development—needed a test-and-development system. The Altair was just the ticket. It was fun to build and get running. *DDJ* was a necessary adjunct to the lab in those days. Not much software was around, and we could always count on the good Doctor for a generous dosage of useful code. Fred has every issue of *DDJ*.

Although Fred is a Forth practitioner now (I'm working on converting him), he got me into C. I was doing a lot of traveling, and one day he shoved a package at me on my way out the door—something to read on the plane, he said. It was the BDS C compiler for CP/M with a copy of Kernighan and Ritchie's *The C Programming Language* thrown in. I remember thinking that a language that only has a single letter for a name must not be much of a language. The names of real languages are healthy, technical acronyms like ALGOL, MACRO-11, and MUMPS. ("MUMPS" is healthy?) I was skeptical about a language with a wimpy name like "C." Nonetheless, I read K&R on the flight, and by the time I "deplaned," I was a C convert. Never had there been such elegance in a programming language. It seemed that all things programmers dislike in code syntaxes were masterfully corrected in the simple and powerful structure of C. Its loose typing created for a perfect match between C and systems programming. C looked like the ideal language. *DDJ* readers don't need me to tell them that, but I like to say it, anyway.

At the first opportunity, I set about to hammer out my first C program, believing with Kernighan and Ritchie that you learn a programming language by writing programs. First, I needed a program to write. As it turned out, I already needed a personal program for one of my silly diversions, those cryptograms that the newspaper runs under the crossword puzzle. You've seen them. They encrypt a clever phrase with simple letter substitutions giving you one of the letters as a clue. You have to decrypt the phrase, usually revealing a groaner of a pun. The hardest part about a cryptogram is mustering the clerical effort to keep track of which letters you've used and where they line up in the message. My daughter Sharon has an eraser that has shredded more paper than Fawn Hall as she tore up the margins of the local daily news decrypting something like "hzsdmpdy nsvlgotrf pm yjr hzsddmpdrf tiddosm npcrt."

So, the first C program I wrote had as its noble purpose the maintenance of letter substitutions and the display of partially decoded cryptograms. It's surprising how quickly you can decode a cryptogram when the tedium is eliminated. Example 1, on page 101, is a PC version of the program. The original version worked with whatever terminal I had at the time. You'll need the ANSI.SYS driver installed to run this version. Most C compilers and interpreters will handle it.

Let's conclude this trip down memory lane with a confession. Despite receiving this column as an assignment, I am not really an expert in the C language. In a moment, I'll describe a problem that took several days

(continued from page 98)

getting solved. I presume to write this column because I am probably just like you—a C programmer who can make programs run and who has a grass-roots understanding of what constitutes a useful, reliable, maintainable C program. If you are looking for the scholarly treatment of the language through an academic microscope, look elsewhere. Those forums exist and they have value. This column is about using the C language as a tool to build programs that people can use.

So much for introductions and the good old days. In the months to come, we'll be indulging a favorite pastime, the publication and explanation of reusable C language tools. I've written four books and several articles on the subject, and the feedback from readers says that the end is well worth its pursuit. This column is a good medium for such material; the publication of free software tools is the honored tradition of *DDJ*.

The little decrypting program in this issue is not necessarily meant to be an example of what you can expect from this column. As with the previous "C Chest" columns, most of these columns will have substantial examples of C code. However, some columns will digress to discuss other matters. I might review a book or a product or interview a C luminary. Occasionally, I will devote a column to the letters from you dear readers. Always, though, there will be at least a vignette of code. Thus the little game in Listing One.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

char crypto [80];          /* encrypted message */
char decoded [80];         /* decrypted message */
char alphabet [] = "\nabcdefghijklmnopqrstuvwxyz";
char subs []       = "                          ";

/* ------ ANSI.SYS screen driver ---------- */
#define cursor(x,y) printf("\033[%02d;%02dH",y,x)
#define clear_screen() puts("\033[2J")

main()
{
    int i, cp, a, b;
    char sb [3];

    clear_screen();
    cursor(1, 6);
    puts("Enter the encrypted quote:\n");
    fgets(crypto, 80, stdin);
    for (i = 0; i < strlen(crypto); i++)
        decoded[i] = ' ';                  /* clear the decrypted msg */
    decoded[i] = '\0';
    while (1)    {
        cursor(1, 9);
        puts(decoded);
        puts(alphabet);                    /* display the alphabet */
        puts(subs);                        /* display the substitutions */
        puts("\nEnter 2 letter substitution: (xy):");
        fgets(sb, 3, stdin);
        if (strncmp(sb, "99", 2) == 0)
            break;
        a = *sb, b = *(sb+1);
        if (isalpha(a) && isalpha(b))    {
            for (cp = 0; cp < 26; cp++)
                if (subs[cp] == a)
                    subs[cp] = ' ';
            subs[b - 'a'] = a;
            for (cp = 0; cp < strlen(crypto); cp++) {
                if (decoded[cp] == b)
                    decoded[cp] = ' ';
                if (tolower(crypto[cp]) == a)
                    decoded[cp] = b;
            }
        }
    }
}
```

Example 1. *PC version of crypto.c*

One of the benefits of writing a column is that you get to plug some of your own stuff. In the past I operated a not-very-successful mail-order business selling software tools. Some of the products are still available, but I have no business interest in them any more. Occasionally this column might discuss one of them if it is relevant to the matter at hand. If that happens, I will tell you of the special relationship that I have with the item so that you might understand my unbridled enthusiasm and take it however you wish. All that aside, I do not hesitate to encourage you to read my books.

My crotchets are based in my opinions and do not reflect anybody's official sanction.

Crotchets

James J. Kilpatrick is a national political commentator with a wide readership on the editorial pages of many newspapers. Besides writing and appearing on television programs, Mr. Kilpatrick has appointed himself one of the guardians of good style and proper usage of the English language. (Self-appointed guardians of style and usage are the only kind we get—no one else cares enough to appoint them.) He has written a wonderful book called *The Writer's Art* in which he lists what he calls his crotchets about writing practices. Kilpatrick's crotchets are pet peeves such as the abuses of the words *hopefully* and *only*, and his list is cleverly presented and makes a lot of sense. If you love to read or write, I recommend that you read *The Writer's Art*. The foreword by William F. Buckley is itself worth the price. Don't be put off if the politics of either author differ

from yours. There are no political statements in *The Writer's Art*, just good sense about the written word. Borrowing Kilpatrick's idea and transferring its application from English to C, each "C Programming" column will offer one or more of my own C programming crotchets when they occur to me and as I come across them. Here are some examples to start the list.

```
        /* --- then ... endif style (K&R, too) ---- */
        if (a == b)     {
             foo();
             bar();
        }
        /* --- begin ... end style --- */
        if (a == b)
        {
             foo();
             bar();
        }
        /* --- do ... doend style --- */
        if (a == b)
             {
             foo();
             bar();
             }
        /* --- poised roadrunner style (ugh!) ---- */
        if (a == b)
        {    foo();
             bar();    }
<end>
```

Example 2: *Several ways how to position braces and indent lines.*

C Crotchet Number 1

I am vexed with C language preprocessor programs or macro files that encourage you to code in some nonstandard syntax, which is then translated into C. The practice suggests that the C language needs improvement (beyond ANSI and C++, of course). For example, some preprocessors or macros want you to use *begin* and *end* in place of the open and close braces, supposedly to make your code look more like Pascal or otherwise more readable. In my opinion, budding C programmers are better off without these so-called C enhancers. It's better to learn the real thing and come to love it a lot sooner.

C Crotchet Number 2

I do not like to read C programs that are inconsistent in their positioning of braces and indenting code. Example 2, this page, presents several ways that you might position braces and indent lines.

Whichever of these or other styles you prefer, be consistent or risk raising the hackles of the next reader of your code. The problem of inconsistency usually sneaks in when someone modifies the code of another and the two programmers prefer different styles. A program with such mixed conventions is a mess.

Two crotchets are enough to give you the idea and should be enough for a month's column. My crotchets are based in my opinions and do not reflect anybody's official sanction. Not all of the crotchets are about C usage. Some might be about compilers or libraries or writers or anything at all that gets the goat. If you wish to add to or subtract from the list—perhaps you grouse at writers who always use *foo* and *bar* in their examples—please send your contributions care of *DDJ* or on BIX (alstevens) on CompuServe (71101,1262). I will include interesting, funny, or otherwise relevant crotchets in this column along with the names of the crotchety contributors.

The Books Department

The book of the month is *Macintosh Programming Secrets* by Scott Knaster. It has no C code, but the book is necessary reading for any programmer who is about to tackle the Macintosh. Don't let the dose of Pascal scare you off; those Pascal guys will come around soon enough, and there are several good C compilers for the Macintosh right now.

Besides being a good introduction to the insides of this fascinating machine, the book is funny just when it needs to be, and is never funny when it does not. I do not usually like computer books that try to be funny. More often than not, the author can't quite pull it off, using tiresome parody or weary satire. Knaster has successfully balanced entertaining humor and worthwhile technical information. Most writers wish they could do that and get away with it. Usually a wise old editor with green eyeshades, a mean red pencil, and a paling sense of humor crosses out all those witty gems. (If you are not reading these words, it has happened again.) That notwithstanding, you are urged to read Knaster's work. He held sway over that editor and managed to get a book to market that informs, entertains, and makes you laugh out loud in places.

Son of K&R

Ten years after its initial publication, *The C Programming Language* has been released in a second edition. It occurs that there might be a lot of C programmers who have never read the first edition. Shame. Plenty of other C books set out to do again what the "white book" does best: describe C to programmers. For my money, K&R is still the best introduction to C if you already know what programming is. The second edition catches up to the emerging ANSI standard (oh, so long it has been "emerging") although Prentice Hall might have been prudent to wait for the official standard rather than promote the book as "based on draft-proposed ANSI C" in a banner on the cover. I suppose the draft is pretty close to the final standard but this early publication *probably* means I'll have to pop for another $21 when the "draft" tag is removed for the next printing of the second edition.

Seen this summer in Paris: a French automobile with a chrome emblem insignia identifying its model: "Turbo

D." A new language and compiler from Philippe, perhaps?

Confusing but Necessary C Constructs

Suppose that you have an array of structures and that each structure has an array of function pointers. Now, given a pointer to the structure array, an integer that subscripts to the desired structure element in the array, and an integer that subscripts to the function you want to call, how do you write the expression that calls the function?

While you think about it, here is some background. This exercise is not another C puzzle, but a real problem encountered in a general-purpose menu driver. The driver uses arrays of structures and strings to describe a hierarchy of menus. This concept of a reusable menu tool is central to most C systems that I develop, and its format has occasionally changed to support different styles of menu presentation. You will not be surprised that the same technique figures significantly in a new "C Programming" project to be revealed soon.

The structure just mentioned describes a menu. The array of those structures represents all the menus in a program. The array of function pointers represents the functions that are to be called when the user chooses the corresponding menu commands. Example 3, this page, shows the pertinent data structures and items. In the problem, the *mnn* pointer points to the menu array, the *curr_menu* integer represents the current menu, and the *selection* integer represents the current selection integer. Imagine that you want to call the proper function, and pass it the menu number and selection number that caused it to be called.

A seasoned C programmer might intuitively come up with the correct expression without thinking about it. A new C programmer, however, is going to have to work on it for a while. You might argue that a new C programmer would never have this dilemma because the construct would not be obvious—the programmer would design a simpler data structure—perhaps distinct arrays for each menu. Nonetheless, these problems are real, and you get a better understanding of the C language after you solve them.

Does the answer seem obvious? If so, congratulations. If not, it will clear up once you understand it. Earlier, as a fledgling C programmer, I designed myself into this corner without realizing how perplexing it was going to be. The correct syntax of the calling statement is what eluded me. So, taking the components of the statement, I tried every variation of parentheses, subscripts, and pointer operators until the statement worked. A solution that worked with one compiler failed with several others. The one shown later was one that all the compilers accepted. To derive it, you can break down the different parts of the problem, a better approach than my stumbling block method. We start by knowing that a function call through a pointer looks like this:

```
struct menus {
    char *menu_selections [MAX_SELECTIONS];
    int (*func[MAX_SELECTIONS])(int, int);
} mn[MAX_MENUS];

struct menus *mnn = mn;
int curr_menu;
int selection;
```

Example 3: *A structure pointing to an array of menus and the functions for controlling them.*


```
(*func)(curr_menu,selection);
```

The *func* pointer contains the address of some function. In the ANSI standard, you can leave out the asterisk and the binding parentheses so that the function call looks like a regular, nonpointer function call. I am not quite ready for this new convention. The older convention—which is still supported, thank goodness—tells you as you read the statement that the call is made through a pointer. This convention gives you a clue as to where to look for the function being called. Such clues are of value when you are searching through some old, perhaps not well-commented code. I prefer it when the language helps to explain itself.

Here is the way that a function's address is assigned to a function pointer:

```
func = funcname;
```

Simple, isn't it? You already knew that. Now suppose that the function pointer is in an array. If the user's menu selection subscripts the array, the assignment looks like this:

```
func[selection] = funcname;
```

This assignment, too, is obvious. But to call through this pointer array, you need a subscripting integer to specify which of the pointers in the array contains the address of the function to be called. The subscripted call looks like this:

```
(*func[selection])
            (curr_menu,selection);
```

This call is not as apparent. I needed some experimenting to arrive at it. Now on to the structure: if the function pointer in the structure was not in an array, and *mm* pointed to the specific structure, the assignment would be like this:

```
mm->func = funcname;
```

Then the call would look like this:

```
(*mm->func)(curr_menu,
                    selection);
```

The progression to this point is also not as obvious, but reasonable. It took me a while to figure out the proximity of the asterisk pointer operator and the parentheses. Now add the selection subscript to the statement and it looks like this:

```
(*mm->func[selection])
            (curr_menu, selection);
```

But in the problem, the *mm* pointer

points to the array—the first structure element in the array—rather than the desired structure element, and the *curr_menu* integer indicates which element in the array represents the current menu. So the final answer is this:

```
(*(mm + curr_menu)–
                >func[selection])
        (curr_menu, selection);
```

The new ANSI rule would simplify it only a little as shown here.

```
(mm + curr_menu)–
                >func[selection]
        (curr_menu, selection);
```

Because of the close relationship between pointers and arrays, there are variations on this expression that will work as well. All of them are equally as cryptic.

These kinds of problems will discourage all but the strongest of hearts. When C programmers gather around the cooler and discuss such matters, they tend to dismiss such solutions as trivial and obvious. It is a maddening characteristic of C that the answers to coding puzzles are elusive when you don't know them and are obvious once you do. This characteristic makes you ask yourself if you are the only one who doesn't understand. Take heart. Unless you are a computer scientist and language expert with a natural propensity for reflexive left-to-right parsing and precedence evaluation, you will have to work out a problem like this one every now and then. By taking the stepwise approach to figuring out a complex C idiom such as is the one illustrated here, you can avoid the trial-and-error method that I innocently used. Here's a tip. Sign on to BIX or CompuServe and ask someone. There are some bright folks there, and almost certainly one of them will have worked out a similar problem. If not, someone is usually willing to go off-line, work out a solution, and sign back on to post an answer. I've never been disappointed by these generous people.

C Programming Project

Over the next several months, I will be using the forum of the "C Programming" column to develop a C language utility program. The full purpose of the program will be revealed in a later column, but for now I will say that the program will be used for modem communications and will run on the IBM PC and compatible computers. Do not dismiss this as just another modem program; I will tell you its full reason for being soon and it will be of interest to most of you. Most of the source code will be published in this column; all of it will be readily available to any *DDJ* reader in printed and magnetic media.

The project will be made up of a number of C tools that are adapted from C libraries, and there will be some custom applications code. I will begin by defining and developing the tools—among them a window library, a menu manager (with that elusive function pointer), a help library, data-entry screens, communications functions, and a text editor. Each month, I will add new tools to the collection. You will be able to use the tools in other applications, but their primary purpose is to support the project. The tool set will evolve in a bottom-up fashion. As I build new capabilities, they will use the functions in the libraries that precede them. Those of you who read the books I write will recognize the style (and some of the functions).

The program will be written in C, of course, to be compiled with Borland's Turbo C, which I selected because of its popularity and its strong text-screen-function library. Turbo C provides PC screen-management functions that lift the burden of worry over display adapters and video retrace signals. Since only the screen-window-management tools will use these Turbo libraries, you can port the rest of the code to other C compilers with a small bit of modification.

This project will be an interactive one. Your comments and suggestions will contribute to its progress. Perhaps between us we'll write the next great American program.

C Programming Sign-off

I would like to close this, my maiden trip in the "C Programming" column, with a tip of the cap to my predecessor, Allen Holub, whose contributions to the advancement of the C language are substantial. We have not met, but I have gained a great deal from his work in his column and in his books. Allen received a vote of appreciation in the preface to the second edition of Kernighan and Ritchie's *The C Programming Language*. In this business, such an acknowledgment is the equivalent of an Academy Award. Fare well, Allen, in your new endeavors.

Availability

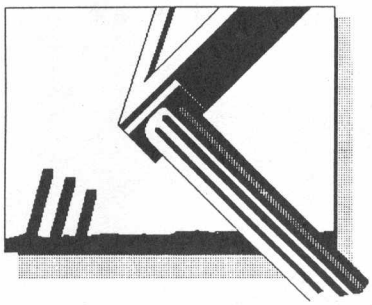
All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

THE FORTH COLUMN

More Real-Time Connections



by Martin Tracy

Loyal readers of this column will have noticed that I often discuss Forth's strong ties to the field of the real-time programming. This month is no exception. In addition to the usual news about chips and language products, I also have some information about the coming Real-Time Programming Conference (formerly known as the National Forth Convention).

In the hardware news this month is Harris Semiconductor's Real Time Express (formerly called the FORCE). The RTX 2000 is a Forth-based 16-bit microprocessor that Harris is billing as "the fastest 16-bit microprocessor available." With sustained performance exceeding 10 MIPS, that claim might not be all hype.

This RISC-like microcontroller, like the Novix NC4016 CPU before it, gets some of its speed by executing the Forth language directly. The chip has both internal parameter and return stacks (keeping the pin count down) as well as a 16 × 16 multiplier, a fast interrupt controller, and three 16-bit timers. The interrupt response time is just 400 nsec. A proprietary ASIC Bus is accessed concurrently with instruction execution and allows for the attachment of external ASICs.

The RTX 2000 is a Harris standard macro cell, that can be extended to a custom VLSI chip. The RTX comes in an 84-lead pin-grid-array package, but will also be available as a ceramic quad package. The price is $190 each in quantities of 1,000. A MIL-Standard 883C version will also be available by the end of the year.

Harris supplies RTX 2000 evaluation boards for less than $1,500, including their RTXDS Software Development System. Alternately, Software Composers plans an IBM PC drop-in board (third quarter 1988) and VME Inc. plans a VMEBus CPU board (first quarter 1989). The Harris RTXDS Software Development System includes a monitor, debugger, and a Laboratory Microsystems Inc. (LMI)-based cross-compiler. By the end of the year, you can expect to see a full Forth Inc. polyFORTH system running on the evaluation board. Both C and ADA language translators are also in the works.

What impressed me most about this chip were the printed specifications. Harris Semiconductor's spec sheet is a 30-page document that includes timing diagrams, electrical specs, opcodes, architecture block diagrams—the works! Call 407-724-7418 for a copy. Harris also has all day seminars and tutorials on the RTX.

As I mentioned earlier, the RTX 2000 is theoretically descended from the original Novix NC4016 CPU, which is still very much alive. You can now buy *More on NC4000*, Volume 7, collected by Dr. C.H. Ting, from Offete Enterprises (1306 S. B St., San Mateo, CA 94402) for $15. Better yet, get his revised and expanded *Footsteps in an Empty Valley*, 3d edition ($25). Dr. Ting's books remain one of the finest sources of information on this chip.

Speaking of the NC4016, Forth Inc. has announced a polyFORTH development system ($2,950) for VME Inc.'s V4000 VMEBus NC4016 CPU card ($2,795). Call them at 213-372-8493 for more information.

And speaking of Forth-based RISC hardware, Johns Hopkins University has been showing off samples from their latest batch of JHU/APL 32-bit Forth chips running at 10 MHz! Mitch Bradley, of Sun computers, talked at the recent annual Rochester Forth Conference about how Sun is using Forth as a machine-independent language for writing drivers for their SPARC Unix systems. The Fujitsu SPARC is a core product of their fifth-generation computer effort. At the same Conference, Dr. Wicks, from Hewlett Packard, showed how a blend of Forth and Lisp is used to program the symbolic math in their calculators.

By the way, the first Australian Forth Conference has evidently been quite successful. I just heard that about 200 people attended it.

LMI's TMS34010 Metacompiler

Laboratory Microsystems announced a version of their metacompiler tar-

Figure 1: *The contest poster for the upcoming Real-Time Programming Convention.*

geted for the Texas Instrument's TMS34010 graphics processor. This chip has a 50-MHz performance with machine-level instructions for both linear and XY-addressed line drawing, pixel block transfer, region fills, clipping, and other graphics operations. With its 16-Mbyte bit-addressed display memory, programming this chip must be quite a challenge.

The metacompiler target was written by Ron Braithwaite and produces a ROMable executable image that may be optionally downloaded to an IBM PC development board. The target sells for $1,000 and is available directly from LMI (213-306-7412).

```
HEX
CODE VIDEO ( cx dx ax -- )    \ IBM BIOS video service
   AX POP  DX POP  CX POP   10 INT   NEXT END-CODE
: TEXT    0 0 2 VIDEO ;    \ Return to text mode.
: GRAPH   0 0 4 VIDEO ;    \ Set high-resolution graphics mode.
CODE PLOT ( x y color -- )
\ Given a coordinate pair and a color code,
\ paint one dot on the screen.
   AX POP  0200 # AX ADD   DX POP  CX POP   10 INT
   NEXT END-CODE
DECIMAL

\ Dr. C. H. Ting's Simplest Line Drawing Algorithm
: draw  ( x1 y1 x2 y2 -- )
\ Draw a straight line between (x1, y1) and (x2,y2).
\ Determine the end condition, where (x1,y1) and (x2,y2)
\ are within one pixel distance.
\ Find the mid point between (x1,y1) and (x2,y2).
\ Insert mid point between 1 and 2, then recurse twice
\ to draw the two segments.
   2over 2over   rot - abs >r
   - abs r> max  2 < if 2drop 3 plot exit then
   2over 2over   rot + 1+ 2/ >r ( y3)
   + 1+ 2/ ( x3) r>  2dup 2rot  recurse  recurse ;

: test1   640 0 do  0 0 i 400  draw  10 +loop ;
: test2   400 0 do  0 0 640 i  draw  10 +loop ;
```

Dr. C. H. Ting's amazing "Simplest Line Drawing Algorithm."

MacForth in the News

MacUser magazine, in an article titled "Picking a Compiler," gave Creative Solution's MacForth a high rating for its toolbox support and dedicated program generation. CSI was seen at MacWorld Expo earlier this year running MacForth under the MultiFinder with color graphics. In addition, Micro Dynamics showed its optical storage WORM drive and DBMS for graphics retrieval. Rumor has it that at least some of its drivers were written in MacForth.

Unicus announced a universal terminal emulator for the Macintosh called MetaTerm. MetaTerm allows users to define the terminal parameters with a command language called ReachForth.

Another terminal support device that has recently come to my attention is the PC Port Controller from Component Systems Inc. (415-861-1345). This box connects ten asynchronous RS232C/RS422 ports together. It includes 192K of buffer memory and a resident Forth running on a 64180 CPU. The proprietary CSI Forth is the control language that specifies the interconnections and buffering. The PC Port Controller can be used as a poor man's local area network, a flexible printer buffer for multiple users, a data concentrator, or as part of an electronic mail system.

Bulletin Boards

Electronic bulletin boards specializing in Forth are still one of the best ways to stay connected with the Forth community. Some of the larger ones are the North Coast Forth Board (612-483-6711), the East Coast Forth Board (703-442-8695), and the Vancouver Forth Board (604-434-5886). Regretfully, the West Coast Forth Board will be closing in June. No matter where you live (well, almost anywhere), the Forth BBS on GEnie is just a local phone call away. Call client services at 800-638-9636 for the number nearest you.

Some of the more interesting files that have appeared on the boards lately include:

- The listing for the source code in Personal Expert System by Townsend and Feucht (ECFB file PXS.ARC)
- Jack Woehr's cross-assembler for the Inmos Transputer (GEnie XASM.ARC)
- A complete BBS that can run in 64K and with one 360K-disk drive (GEnie ARIEL.ARC)
- A Prolog implementation for the Laxen and Perry F83 dialect (GEnie PROLOG2.ARC)
- A JForth version of Bob La Quey's neural network simulator (GEnie J4NEURAL.ARC)

The East Coast Forth Board recently posted a Forth bibliography. Don Madson has compiled a comprehensive list of Forth books in the Library of Congress. You can download the list from the ECFB by reading messages 463–465.

1987 FORML Conference Proceedings

The combined proceedings of the 1987 9th Annual FORML Conference and the euroFORML 87 Conference are now available from the Forth Interest Group (408-277-0668). The conference itself was reviewed in an earlier column, and it was great. Regrettably, Dr. Ting's amazing "Simplest Line Drawing Algorithm" was lost somewhere in the shuffle, so I

am reprinting it in this column (see page 116). This recursive routine draws a line by cutting it artfully in two until the endpoints lie in adjacent pixels.

Call for Presentations

This is a call for presentations for the Real-Time Programming Convention, November 18–19, at the Grand Hotel in Anaheim, Calif. This event, formerly called the Forth Convention, is sponsored by the Forth Interest Group. This year's opening speaker is Ray Duncan, a well-known authority on IBM PC programming and known to *DDJ* readers as well for his "16-bit Toolbox" column. The banquet speaker is none other than Jef Raskin, head of the original Macintosh development team and inventor of the Canon Cat.

This year the conference theme is Real-Time Programming Systems, so if you work with the real world and in real time, this is the place to be. This conference is not limited to members of the Forth community, so whether you use Forth or machine code or C, if you are performance-oriented and program "down to the metal," if you use A/D's and D/A's on single-board computers, if you need the speed of language-oriented RISC machines, come to this conference.

We need presentations in the following areas:

Real-time operating systems
Language-oriented RISC machines
Parallel processing
Languages for data acquisition and analysis
Intelligent instrumentation
Working neural nets
Adaptive devices
Software peripheral controllers
Applications in
- Aerospace
- Medicine
- Laboratories
- Machine vision
- Digital signal processing
- Robotics
- Automation

By presentations we mean talks or demonstrations, preferably with an audio-visual component.

Of course, you will see and hear a lot of Forth at this convention. After all, Forth is one of the very best solutions for real-time programming. So come and meet your vendor or hear Chuck Moore's fireside chat and see what the real-time world has to offer.

Programming Contest

Better yet, come and see or participate in the first Fastest Programmer in the World Contest. This contest is open to any programmer using any computer, any software. See the down-sized poster on page 114 for details. We anticipate a large response, so there will be some sort of screening before the contest. Write for rules to the Programmer's Contest, c/o the Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.

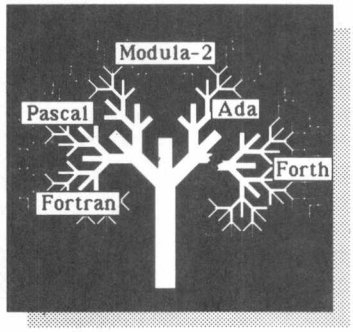
Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Improving on DIR



One of the first things a new user to DOS learns is how to use the *DIR* command. And as soon as the Wonder Of It All starts to lose its magical shimmer, he or she begins to chafe against its inadequacies: "One of these days, I'll write something better." Well, after five years of such grumbling, I finally got around to writing a better program, which I will share with you in this month's column. Along the way, you'll see the way to do some system-level programming with Turbo Pascal 4.0.

There are many complaints about *DIR*. Here are my favorites:

- *DIR* has to be told with the /P modifier not to scroll
- *DIR* doesn't show file attributes
- *DIR* doesn't list system files and hidden files, making them almost impossible to detect
- The size that *DIR* shows in bytes is not the amount of space that the file actually occupies on disk
- *DIR* shows free space on the disk, but not the total space that the listed files claim
- *DIR* does not show how big the disk actually is and what percentage of the disk space is currently occupied by files

There are a number of everyday situations in which *DIR* doesn't quite do the job. For example, say you want to copy a subdirectory to a floppy disk. But how many disks do you need? With *DIR* you have to use a calculator to add up the space. That's dumb: that's the sort of work computers are meant to do. Besides, a file almost always requires more space than its number of bytes, so even a manual computation is a guess. The same principle holds true if you want to copy all your .PAS files from a directory.

by Kent Porter

Another example of the problems with *DIR* occurs when you're trying to get rid of a directory. You must first delete all the files, then go to the next higher level, and remove the directory with *RmDir*. However, *DEL* doesn't delete hidden, system, or read-only files, and *DIR* doesn't even let you know that the first two types of files exist. Thus, *RmDir* refuses to cooperate, and you haven't a clue as to the reason. You'd know the reason if you could see all the files and their attributes.

And finally, it's good to know how full your disk is, rather than how many bytes are free. You can set an arbitrary threshold, such as 90 percent, at which point you need to consider cleaning house.

Enter the improved *DIR*. I call it *SUB* because it's chiefly a subdirectory listing tool. *SUB* accepts either no command-line argument (meaning "list all files in the current directory"), or *DIR*-style wildcards. Table 1, on page 128 presents the command-line parameters for *SUB*.

So far *SUB* should seem very familiar to you, because these arguments are exactly the same ones that *DIR* expects. The difference is not in the input—the difference is in the output. Figure 1, page 128, shows the results of the command *SUB \ .COM* on my hard disk. The command lists the five .COM files in the root directory. The display is quite different from what you get by using the equivalent *DIR* command. *DIR* would not even show the first two files, which have the attributes System and Hidden. Furthermore, *DIR* does not reveal that OPTIMIZE.COM is a read-only file, which *SUB* does.

Perhaps the most important difference between the *SUB* output and the typical *DIR* output is in the last two lines. They show that the five files consist of a total of 86,045 bytes in 86K of disk space, and that the disk is 39.02 percent occupied.

The total of 86,045 bytes in 86K is unusual. A disk is divided into fixed-size allocation clusters. The number of kilobytes of occupied disk is ordinarily much higher than the number of bytes. For example, my hard disk's root directory contains 10 .BAT files with a total of 557 bytes, but the files occupy 20K.

What is the reason for this difference? The minimum space given to a file of one byte or greater is one allocation cluster. (Files of 0 bytes and volume labels, which are purely directory entries, claim no disk space.) If a file consists of one byte to 2048 bytes, it claims a 2K allocation cluster. (I'll discuss cluster sizes later in this column.) A file of 2049 bytes is 1 byte greater than the cluster size, so the file requires two clusters, or 4K. In 10 files, 557 bytes is an average

of 55.7 bytes per file, yet each file takes up the minimal cluster of 2K. This size requirement is the reason that the 10 small .BAT files in my root directory use up 20K.

All this information is important in determining how many floppy disks you need to back up a set of files. Suppose you have 180 files of 1 byte each. The total data space is 180 bytes, but the total file space is 360K, or another floppy disk. This difference between data space and file space is the reason for wanting to display not only total bytes, but total kilobytes used.

The top of the *SUB* output contains the line

Directory for <mask> in <path>:

which indicates what is being listed. The next line gives labels for each column. The next possible 21 lines show actual directory entries satisfying <mask> in <path>. If there are *less than 21 entries*, as in the case of Figure 1, you will see the two summarization lines. If there are 21 entries or not, *SUB* skips a line and displays the prompt

– MORE –

You simply press a key to advance to the next panel.

As with *DIR*, you can use *SUB* to examine the directories of disks other than the default. For example, to see all the files in the root directory of the diskette in drive B, type

SUB B:

The utility "signs on" to drive B, does its thing, and then returns at the DOS prompt level to the originating drive and directory. The same process occurs when you list on the default drive a directory that is other than the current one.

Now look at the flow of the overall SUB.PAS program shown in Listing One (beginning on page 142). As with all structured programs, you can understand the logic most quickly if you work down from the highest level: the body of the main program.

Because of the many forms that the command-line argument can have, the program makes up to four attempts to process the argument using different interpretations. A successful attempt sets the Boolean variable *thru* to *TRUE*, and bypasses further attempts. If no effort is successful, *thru* is still *FALSE* at the end, so the program prints the message

PATH NOT FOUND

and quits.

Each attempt differs from the others in its set-up. The reason for this difference is to accommodate the possible interpretations of the command-line argument. The middle two attempts try to change directories using the *ChDir* procedure, which is similar to the DOS command. Turbo's *ChDir* generates a run-time error when the path is not found; to get around this, the program uses the {*$I–*} pragma to disable run-time error checking, and then detects whether the procedure succeeded by checking Turbo's *IOResult* variable. A nonzero result indicates failure.

The third attempt calls the local procedure *Separate*. This procedure uses substring manipulation to break an argument such as *\DOS*.COM* into its path and its file mask components (*\DOS* and **.COM*, respectively).

All of the attempts call a procedure named *ListFiles*. It contains nested procedures for display control. The most interesting feature of *ListFiles* is, in fact, the basis for obtaining directory entries from within a program: the Turbo procedures *FindFirst* and *FindNext*. They're syntactic sugar coatings for DOS functions *4Eh* and *4Fh*.

A call to *FindFirst* loads file and attribute masks into a data structure of type *SearchRec* (defined in Turbo's DOS unit), and then initiates a search of the current directory for a file that matches the masks. If successful, the system global variable *DosError* is set to 0 and the *SearchRec* structure contains information about the file. Subsequent searches for other files matching the same masks are made with calls to *FindNext*. You can keep calling *FindNext* to get successive files, until *DosError* changes to nonzero. This change indicates that no files remain that satisfy the mask in the directory.

Let's look at the *WriteFileInfo* procedure, which *ListFiles* calls for each hit made by *FindFirst/FindNext*. We'll discuss file attributes a little later.

WriteFileInfo displays information contained in the *SearchRec* structure. Three nested functions do most of the work for *WriteFileInfo*.

The *SearchRec Time* field is a zoned 32-bit integer that the *TimeStamp* function converts into a formatted string. This string shows the date and time of the file's most-recent update: a lot of string-fiddling, but not especially opaque.

The *SizeInK* function rounds the number of bytes in the file up to the next-higher multiple of the disk's allocation cluster size. *SizeInk* then converts the number into kilobytes by dividing by 1024. The allocation cluster size is in the variable *BlockSize*, which is loaded by using a DOS call in the *AllocInfo* function elsewhere in the program. This size varies according to the media; for many disks, the size is 2K, but it can range from 512 bytes to 4K. [For more details on cluster sizes, check Ray Duncan's *Advanced MS-DOS*. —ed.]

Now let's turn to file attributes. Every DOS directory entry contains a field called the attribute byte. The field is a catch-all for storing status, access, and visibility information. The low six bits are all significant and are listed in Table 2 (on page 128).

The archive bit is sometimes called the "dirty bit." This bit goes to 1 when the file is updated. Programs such as FASTBACK and the DOS BACKUP utility reset this bit for each file copied, and by checking it during subsequent runs, they determine whether or not the file needs to be backed up again.

In the DOS unit, Turbo furnishes named constants for these bits: *ReadOnly*, *Sysfile*, and so forth. The *Attribs* function in *WriteFileInfo* uses these constants to test bits and to build a displayable string that shows the file's attributes. Turbo also furnishes the constant *AnyFile*, which has all six low-order bits turned on.

This constant serves as the attribute mask for the invocation of *FindFirst/FindNext*: it tells them to select any file satisfying the name mask, regardless of its attributes.

And that's how *SUB* does it.

I find this *SUB* command much more useful than *DIR*, and you probably will, too. Place *SUB* in your *\DOS* directory or somewhere along the chain established by the *PATH* command. After you do this, you can execute it no matter where you are on the hard disk.

What's In a Directory Entry?

Turbo Pascal's *FindFirst* and *FindNext* procedures extract information from directory entries, which are the basic housekeeping structures for disk management. A DOS disk has two kinds of directories; the root is a fixed-size area on the first few tracks of the disk, while a subdirectory is a special kind of variable-sized file that contains control information about other files. Both a root directory and a subdirectory consist of 32-byte data structures.

Expressed in Pascal notation, the format of a directory entry is:

```
TYPE DirEnt = RECORD
  Filename : ARRAY [0..7] OF CHAR;
  Extension : ARRAY [0..2] OF CHAR;
  Attribute : BYTE;
  Reserved : ARRAY [0..9] OF BYTE;
  Timestamp,
  Datestamp : WORD;
  FATentry : WORD;
  Filesize : INTEGER;
END;
```

The FAT entry points to the location with the File Allocation Table where the allocation chain begins. FAT entries map one-for-one to allocation clusters.

Consequently, this is all the information DOS needs to know about any given file. A program such as *SUB* doesn't need all this information; the reserved area and the FAT entry are of no practical value to most applications. For this reason, *FindFirst* and *FindNext* copy the other fields to the *SearchRec* structure. — K.P.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext.221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 142.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

.#	Same as *.#
\DIR	Everything in another directory
\DIR\.#	Specified files in another directory
\DIR\#.#	Ditto
\DIR\SUBDIR\...	Same for longer paths
\DIR\SUBDIR\...\.#	
\DIR\SUBDIR\...\#.#	
SUBDIR	Everything in a lower subdirectory
SUBDIR\.#	Same as above, lower subdirectories
SUBDIR\#.#	
SUBDIR\.#, and so on.	

The symbol # denotes any combination of wildcards and literal characters forming a filename mask. For example, either SUB \DOS\.COM or SUB \DOS*.COM lists all .COM files in the \DOS directory. Similarly, SUB \LOTUS\AC*.WK? lists all .WKS and .WK1 files beginning with the letters AC in the \LOTUS directory.

Table 1: *Command-line parameters for SUB.*

Directory for *.COM in \ :

Name	Date	Time	Attrib	Bytes	Size
IBMBIO.COM	12-30-85	12:00P	...SH.	16369	16K
IBMDOS.COM	12-30-85	12:00P	...SH.	28477	28K
COMMAND.COM	12-30-85	12:00P	A.....	23791	24K
OPTIMIZE.COM	10-14-87	06:55P	A....R	1968	2K
QUICKBUF2.COM	12-18-87	10:02A	A.....	15440	16K

5 files, 86045 bytes, 86K space
13039616 bytes free out of 33419264 total (39.02 % utilization)

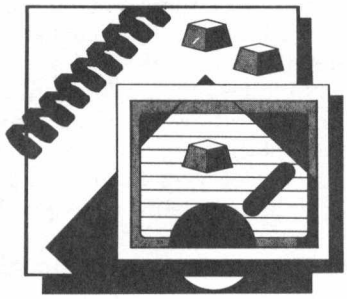
Figure 1: *Example SUB output for a root directory.*

Bit	Means
0	Read-only
1	Hidden
2	System
3	Volume label
4	Directory
5	Archive

Table 2: *Lower six bits of the attribute byte of a directory entry.*

Do They Really Pay For Every Line?

— Rickie Lee Jones, Magazine.



On the Rickie Lee Jones album *Magazine*, a song called "Gravity" is preceded by an instrumental track called "Prelude to Gravity." I really believe that an understanding of alternative programming paradigms is so gravely important to American software developers today that this column could be called the gravity track of this magazine. But so far I've only been playing the prelude. To really develop the theme, I'll need to bring in other voices. This month I'll wrap up this number and set the stage for the polyphonic theme to come.

First, though, I'll beat this old drum once again. Certain facts are gathering to cast an ominous shadow on American software development.

1. Widely known techniques exist today for decomposing large problems into small, discrete tasks that can be handed to a shop full of coders.
2. Performance is rarely as important to a software client as prompt delivery.
3. and 4. You do not have to understand English to write code, and no shipping cost is in effect for software that is written offshore. It seems inescapable that the American coder will increasingly be in competition for work with Third World talent. Any programmer whose work is primarily coding has as much reason for concern about the financial future as a Detroit auto worker.

Nobody wants to be the victim of change or to be on the wrong side of the door when it closes. Diversification and abstraction are among the few strategies that can be effective when you are up against uncertainty. Don't put all your eggs in one basket. Concentrate on the principle behind the procedure. If you can't find the answer, restate the question. Take the paradigmatic view. I believe that the American programmers who prosper in the 1990s will be those who can think in several programming paradigms.

by Mike Swaine

This column has thus far proselytized for the paradigmatic view and danced around the edges of the issue. This month I present some parallel algorithms that I hope will be of interest. Next month will see the beginning of a series of discussions with, and summaries of, lectures by the experts in object-oriented design, Lisp and Prolog development, parallel processing, Risc-centered design, and other paradigms of programming.

Parallel Algorithms

The difficulty in all parallel architectures is getting the correct data to the correct place at the correct time. Performing the computation is the easy part.

Tom Knight of MIT, quoted in Tony Durham's *Computing Horizons*, said it's not just the architectures. Glancing at the tables of contents of some of the books on parallel programming, you can see that communication is a deep concern in parallel algorithms.

This month I'll present some elementary practical algorithms from the growing literature on parallel processing. I'll touch on some of the things that people who work with parallel algorithms have to say about the potential and present performance of these algorithms. Some of this discussion will doubtlessly be familiar, but it may be helpful to those who have limited experience with parallel algorithms. Also, the work of Ewald Speckenmeyer, et al. on superlinearity may be new to many readers.

Sorting in Parallel

With sobs and tears he sorted out
Those of the largest size,
Holding his pocket handkerchief
Before his streaming eyes.

—Lewis Carroll,
Through the Looking-Glass

Sorting is one of the most analyzed of computational problems. The first serious algorithm learned by a computer science student (with sobs and tears, as often as not) is likely to be a sorting algorithm. The working programmer is as likely to know, offhand, the performance characteristics for common sorting algorithms as for any algorithms. In his well-written book *Algorithmics: the Spirit of Computing*, David Harel presents parallel versions of a simple Mergesort (see Example 1, page 132) and discusses the theoretical performance of some sequential and parallel sort algorithms. Table 1, on page 132, is based on his Figure 10.2, with some additions.

The figures for the product of the number of processors and the worst-case time are an important measure for assessing parallel algorithms. To some extent, you can trade off size and time in parallel implementations: in other words, buying more speed with more processors. The product figure thus gives a measure of overall efficiency, and it bears a nice relationship with the worst-case time figure for sequential algorithms. Any parallel algorithm can be sequentialized, with a total time of the order of magnitude of the sum of the times taken by all the processors in the parallel version. So, the product measure for a parallel algorithm cannot be better than the lower bound

(continued from page 130)

on the problem's sequential-time complexity. A linear increase in speed with additional processors is the best you can expect, though communication overhead in the parallel implementation will normally result in something less than a linearity. For sorting algorithms, the lower bound on the problem's sequential-time complexity is *N* log *N*, so that also bounds the product measure for a parallel sort.

Harel presents an odd-even sorting network algorithm and Sara Baase (in the second edition of her *Computer Algorithms: Introduction to Design and Analysis*) presents a parallel mergesort algorithm (Example 2, page 134). Both of these algorithms approach this *N* log *N* bound, with *N* processors and O((log*N*)^2) time. Both have small "big-O" constants. Harel also discusses an "optimal" parallel sorting algorithm that breaks this *N* log *N* down into an order-*N* processor demand and worst-case time on the order of log*N*. This Harel algorithm is based on a sorting network, but it is unfortunately impractical for

```
function parallel-sort L
        if length(L)>1
        then
                    split L into L1 and L2
                    par
                                parallel-sort L1
                                parallel-sort L2
                    merge L1 and L2 into L
        end if
        return L
end function
```

Example 1: *Pseudocode representation of a parallelized Mergesort algorithm, based on Harel, p. 261. It sorts a list L of length N in O(N) time, using N processors. The recursive calls are executed in parallel. It is a straightforward parallelization of a sequential algorithm, and not a particularly good one, having a product measure is O(N2) (see Table 1).*

Algorithm	Size	Time	Product
Sequential sorting algorithms:			
Bubblesort	1	O (N^2)	O (N^2)
Mergesort	1	O (N logN)	O (N logN)
Heapsort	1	O (N logN)	O (N logN)
Quicksort	1	O (N logN)	O (N logN)
Parallel sorting algorithms:			
Parallelized Mergesort (Fig. 1)	O (N)	O (N)	O (N^2)
Odd-even sorting network	O (N(logN)^2)	O ((logN)^2)	O (N(logN)^4)
Iterative Mergesort (Fig. 3)	O (N)	O ((logN)^2)	O (N(logN)^2)
'Optimal' sorting network	O (N)	O (logN)	O (N logN)

Table 1: *Performance of some sequential and parallel sorting algorithms (after Harel). The Product column is the product of the Size (number of processors) column and the Time (worst-case performance) column. This product measure provides a way of comparing the performance of parallel and sequential algorithms.*

(continued from page 132

normal purposes because of outrageously large "big-O" constants.

Restaurant Algorithms

I got sick and tired of having to decide what kind of dessert I was going to have at the restaurant, so I decided it would always be chocolate ice cream, and never worried about it again — I had the solution to that problem.

— Richard Feinman,
Surely You're Joking, Mr. Feinman!

It doesn't seem realistic that the development of parallel algorithms can rest entirely on parallelization of sequential algorithms like Mergesort. In some cases you may have to look at the problem anew.

Sometimes, that can mean changing the very question you are asking. Relaxing the pure algorithmic requirement of a guaranteed solution, or a solution guaranteed in a finite time, can radically change the search strategy. What you end up with is a solution to a different problem. Heuristics and probabilistic techniques can play a big part in parallel algorithms.

Number of Processors	2	4	8
Average speedup	2.46	4.68	9.59

Table 2: *Superlinear performance in a parallel algorithm. Speckenmeyer et al., cite these results as evidence for greater-than-linear speedup with multiple processors.*

```
for pass=1 to ceiling(lgN) do
        k:=2pass-1
        par
                    merge length-k sublists Li and Li+1
                    merge length-k sublists Li+2 and Li+3
                    {etc., merging adjacent sublists pairwise}
end for
```

Example 2: *Sketch of an iterative parallel Mergesort algorithm, after Baase, p. 376. It sorts a list L of length N in O((logN)²) time, using N processors. The recursion of the algorithm in Figure 1 is unwound here, with all length-1 sublists meged first, then all length-2 lists, then all length-4 lists, and so forth. The word "ceiling" refers to the ceiling function: ceiling(X) is the least integer greater than or equal to X.*

A classic problem in the allocation of resources is the problem of the "Dining Philosophers." In one version of this problem, *N* plates are set at a round dinner table. A single chopstick is placed between each pair of adjacent plates, so that it would be within the reach of a person seated before either of the two plates. On each plate is some rice. Don't ask how much rice; all you could ever want, a veritable mountain of rice. We are given to understand that this is an all-you-can-eat Chinese restaurant. At unpredictable intervals, any one of *N* philosophers will wander into this restaurant, sit down in an available chair, attempt to eat some rice from the plate, and equally unpredictably get up and leave the room, only to repeat this performance as unpredictably as before. You've probably seen the situation hundreds of times.

Clearly, the management has erred in supplying only *N* chopsticks for *N* philosophers; this is a recipe for deadlock. As most Chinese restaurateurs realize, if all *N* philosophers came to the table at once, they would all grab for chopsticks at once and they would all starve to death. The problem is to prevent the tragic spectacle of philosophers starving while their plates are laden with rice, and it can be shown that this "Dining Philosophers problem" does not admit of a fully distributed, fully symmetrical solution (unless the restaurant hires a doorman to keep the room's philosopher density down to *N*-1.)

A coin toss introduces an element of randomness that guarantees that the probability of a true deadlock is zero.

"Fully distributed" means no shared memory. The philosophers' protocols may use only distributed information: "Fully symmetric" means that all their protocols are identical. These are the kinds of constraints that can be expected from parallel processing and in select restaurants.

The philosophers will eventually starve unless...

Harel's presentation of the algorithm that saves the philosophers is lucid. Example 3, page 138, shows the protocol he prescribes for each philosopher. Note how he bends the constraint that the philosophers' protocols be identical. What they do is identical; what results from their actions may differ. A coin toss introduces an element of randomness that guarantees that the probability of a true deadlock is zero.

The "Dining Philosophers" proto-

(continued from page 135)

col is not new, nor is the use of random information to improve the performance of a deterministic algorithm. A number of currently hot topics in parallel processing incorporate this simple insight. Simulated annealing, the Metropolis algorithm, neural networks, Boltzmann machines, and Parallel Distributed Processing are among the heuristic-based techniques that have attracted a legion of algorists as motley as the father of the H-bomb and the codiscoverer of the genetic code.

And it looks like the route to superlinearity.

Superlinearity

I suppose Siegel and Shuster won't get any royalties from this, either.

— attributed to Jerry Pournelle.

If one processor is good, wouldn't two processors be 2.46 times as good? Three West German researchers say "yes." Ewald Speckenmeyer, Burk-

```
repeat forever
        carry out private activities until hungry
        repeat until sated
                    toss coin
                    if heads
                    then
                                firstSide:=left
                                secondSide:=right
                    else
                                firstSide:=right
                                secondSide:=left
                    wait until chopstick on firstSide is
                                                available
                    lift chopstick on firstSide
                    if chopstick on secondSide is not available
                    then
                                put down chopstick on firstSide
                    else
                                lift chopstick on firstSide
                                eat until sated
                                put down chopstick on firstSide
                                put down chopstick on secondSide
                    end if
        end repeat
end repeat
```

Example 3: *Protocol for the "Dining Philosophers" (based on Harel, pp. 303-304). The algorithm is deadlock-free with probability zero, but only if the philosophers decide randomly which chopstick to pick up first. If they leave out the coin flip, they will eventually deadlock and starve.*

```
push input formula F
repeat until stack is empty or satisfiability is reported
        pop formula F
        if F=0
        then
                    report "satisfiable"
        else
                    choose "appropriate" variable X
                    split F into FX and FX'
                    for e in {X,X'} do
                                if the empty clause is not in Fe
                                then
                                            push Fe
                                end if
                    end do
        end if
end repeat
```

Example 4: *Sequential backtracking algorithm of Speckenmeyer et al. The algorithm determines the satisfiability of a formula. Some details (such as how an "appropriate" X is selected and how formulas and clauses are represented and what constitutes satisfiability) are suppressed, but to see what the authors parallelized it is only necessary to focus on the for . . do loop. F is split into two parts, and, in effect, the pushes get done in parallel via separate processors.*

PROGRAMMING PARADIGMS
(continued from page 138)

hard Monien, and Oliver Vornberger have used probabilistic information to create a parallel algorithm that, they claim, achieves superlinear speedup. They got the speedup results listed in Table 2, page 134, in actual tests. They did not get these results by designing a new heuristic that works well in a parallel implementation, but poorly in a sequential one. They parallelized an existing sequential algorithm known to work well in the case of a single processor.

The algorithm, discussed in *Supercomputing 1987*, Houstis, et al., eds., is a backtracking algorithm for determining the satisfiability of a formula. A simple sequential version of the algorithm is shown in Example 4, below, along with a discussion of the way in which the authors parallelized it. The authors maintain that, although the parallel algorithm could be simulated on a sequential machine, this would not necessarily lead to an improved sequential algorithm because of overhead costs.

The parallel version of the algorithm is able to break the rules and achieve superlinearity, the authors say, because solutions are distributed nonuniformly on the average. This is apparently the first time that the nonuniformity of the solution density has been used in analyzing an algorithm. If the authors are right that the observed superlinearity is because of the nonuniformity in solution density, then they must also be right that any problem exhibiting such solution density nonuniformity could yield to a superlinear parallelization. Does this suggest an entirely new approach to difficult problems: start by attempting to characterize the solution space? Still unknown: how large is the class of problems that give superlinear speedup?

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). **DDJ**

Vote for you favorite feature/article.
Circle Reader Service **No. 8.**

STRUCTURED PROGRAMMING

Listing One *(Text begins on page 120.)*

```
PROGRAM sub;

{ Enhanced version of DIR command }
{ Turbo Pascal 4.0 }
{ K. Porter, DDJ, August 88 }

USES dos, crt;

TYPE  maskType = STRING [12];
      pathType = STRING [80];

VAR   oldDir, searchPath : pathType;
      fileMask           : maskType;
      BlockSize, Nfiles  : INTEGER;
      TotalBytes, TotalK : LONGINT;
      Thru               : BOOLEAN;

{ ------------------------------------------------------------ }

PROCEDURE WriteFileInfo (VAR files : SearchRec);

     { List filename, date, time, etc. }

VAR   KBytes : LONGINT;

  FUNCTION TimeStamp : STRING;

     { Return file date/time by unpacking time field }

  VAR  stamp             : DateTime;
       StampStr, WorkStr : STRING [20];
       ap                : CHAR;

  BEGIN
    UnpackTime (files.time, stamp);

    Str (stamp.month, StampStr);          { Format year as string }
    IF stamp.month < 10 THEN StampStr := '0' + StampStr;
    Str (stamp.day, WorkStr);
    IF stamp.day < 10 THEN
      StampStr := StampStr + '-0' + WorkStr
    ELSE
      StampStr := StampStr + '-'  + WorkStr;
    Str (stamp.year - 1900, WorkStr);
    StampStr := StampStr + '-' + WorkStr + '    ';

    IF stamp.hour > 11 THEN
      ap := 'p'
    ELSE
      ap := 'a';
    IF stamp.hour > 12 THEN
      stamp.hour := stamp.hour - 12;
    Str (stamp.hour, WorkStr);                { Format time string }
    IF stamp.hour < 10 THEN
      StampStr := StampStr + '0' + WorkStr
    ELSE
      StampStr := StampStr + WorkStr;
    Str (stamp.min, WorkStr);
    IF stamp.min < 10 THEN
      StampStr := StampStr + ':0'
    ELSE
      StampStr := StampStr + ':';
    TimeStamp := StampStr + WorkStr + ap;
  END;


  FUNCTION Attribs : STRING;

     { Return file attributes as a string of indicators }

  VAR   attrib : STRING [6];

  BEGIN
    attrib := '......';
    WITH files DO BEGIN
      IF attr AND ReadOnly  <> 0 THEN attrib [6] := 'R';
      IF attr AND Hidden    <> 0 THEN attrib [5] := 'H';
      IF attr AND Sysfile   <> 0 THEN attrib [4] := 'S';
      IF attr AND VolumeID  <> 0 THEN attrib [3] := 'V';
      IF attr AND Directory <> 0 THEN attrib [2] := 'D';
      IF attr AND Archive   <> 0 THEN attrib [1] := 'A';
    END;
    Attribs := attrib;
  END;
  FUNCTION SizeInK : INTEGER;

     { Return allocated size of file in K }

  VAR  size : LONGINT;
```

(continued on page 144)

Listing One *(Listing continued, text begins on page 120.)*

```
  BEGIN
    IF files.size = 0 THEN
      SizeInK := 0
    ELSE BEGIN
      Size := files.size DIV BlockSize;
      IF size MOD BlockSize <> 0 THEN
        Inc (size);
      IF size = 0 THEN size := 1;
      SizeInK := (size * BlockSize) DIV 1024;
    END;
  END;


BEGIN  { Body of WriteFileInfo }
  Write   (files.name);     Gotoxy (21, whereY);
  Write   (TimeStamp);      Gotoxy (42, whereY);
  Write   (Attribs);        Gotoxy (53, whereY);
  Write   (files.size : 6); Gotoxy (64, whereY);
  KBytes := SizeInK;
  Writeln (KBytes : 3, 'K');

  TotalK := TotalK + KBytes;                { Accumulate totals }
  TotalBytes := TotalBytes + files.size;
  Inc (NFiles);
END;
{ ------------------------------------------------------------- }

PROCEDURE ListFiles (path : pathType; mask : maskType);

    { List files in currently selected directory using mask }


VAR  files     : SearchRec;
     heading   : STRING [160];
     lineCount : INTEGER;


  PROCEDURE StartPage;                      { Begin a new page }

  BEGIN
    ClrScr;
  Writeln (heading);
  Write   ('Name                Date        Time     Attrib');
  Writeln ('      Bytes     Size');
  LineCount := 3;
END;


PROCEDURE CountLines;   { Count lines, start new page if full }

VAR   ch : CHAR;

BEGIN
  Inc (LineCount);
  IF LineCount = 24 THEN BEGIN
    Gotoxy (1, 25);
    Write ('-- MORE --');
    ch := ReadKey;
    StartPage;
  END
END;
PROCEDURE ShowTotals;       { Show total bytes, K, files, etc. }

VAR   free, size : REAL;

BEGIN
  size := DiskSize(0);         { Size of default disk in bytes }
  free := DiskFree(0);
  Write   (NFiles, ' files, ', TotalBytes, ' bytes, ');
  Writeln (TotalK, 'K space');
  Write   (free:1:0, ' bytes free out of ', size:1:0, '
           total (');
  Write   ((((size-free) / size) * 100.0) : 5 : 2);
  Writeln ('% utilization)');
END;


BEGIN  { Body of ListFiles }
  Heading := 'Directory for ' + mask + ' in ' +
             path + ':';
  StartPage;
  FindFirst (mask, AnyFile, files);
  IF DosError = 0 THEN
    REPEAT
      WriteFileInfo (files);
      CountLines;
      FindNext (files);
    UNTIL DosError <> 0;
  ShowTotals;
END;
{ ------------------------- }

PROCEDURE Separate (VAR path : pathType; VAR mask : maskType);

    { Break out path and mask from command-line argument }

VAR   p, c : INTEGER;

BEGIN
  path := ParamStr(1);
  mask := '';
  p    := Length (path);
  WHILE (path [p] <> '\') AND (p > 0) DO   { Find last \ in arg }
    Dec (p);
  IF p > 0 THEN BEGIN
    FOR c := (p + 1) TO Length (path) DO
      mask := mask + path [c];                    { copy file mask }
    IF p = 1 THEN
      path [0] := chr (1)                         { backslash only }
    ELSE
      path [0] := chr (p - 1);          { truncate path before \ }
  END;
END;
{ ------------------------- }

FUNCTION AllocInfo : INTEGER;

    { Returns the size of a disk allocation unit in bytes }

VAR   reg : registers;

BEGIN
  reg.ah := $1B;
  Intr ($21, reg);                          { DOS Int 21h, Fcn 1B }
  AllocInfo := reg.al * reg.cx;         { sec/cluster * sec size }
END;
{ ------------------------- }

PROCEDURE Check (VAR mask : maskType);

    { Check file mask, complete if necessary }

BEGIN
  IF mask [1] = '.' THEN                  { if form is '.EXT'... }
    mask := '*' + mask;
  IF pos ('.', mask) = 0 THEN         { if mask has no period... }
    mask := mask + '.*';
END;
{ ------------------------- }

BEGIN   { Main program }
  GetDir (0, oldDir);               { Save the current directory }
  BlockSize  := AllocInfo;                  { Initialize globals }
  TotalBytes := 0;
  TotalK     := 0;
  NFiles     := 0;
  Thru       := FALSE;
  fileMask   := '*.*';                { Initialize mask and path }
  searchPath := oldDir;

  IF ParamCount < 1 THEN               { No command-line arg, so }
    BEGIN                            { show all files in curr dir }
      ListFiles (searchPath, fileMask);
      Thru := TRUE;
    END;

  {$I-}                              { Disable auto error checking }
  IF not thru THEN BEGIN  { Is command SUB <dir> or SUB \<dir>? }
    searchPath := ParamStr(1);
    ChDir (searchPath);                   { Try to set directory }
    IF IOResult = 0 THEN BEGIN              { List if successful }
      {$I+}                           { Re-enable error checking }
    ListFiles (searchPath, fileMask);
    Thru := TRUE;
  END;
END;
 {$I-}
 IF not thru THEN BEGIN             { Is command SUB <dir\*.*>? }
   Separate (searchPath, fileMask);
   ChDir (searchPath);                  { Try to set directory }
   IF IOResult = 0 THEN
     IF Length (FileMask) > 0 THEN BEGIN
       Check (fileMask);
       ListFiles (searchPath, fileMask);
       Thru := TRUE;
     END;
 END;

 IF not thru THEN BEGIN                     { Is command SUB *.*? }
   fileMask   := ParamStr(1);
   Check (fileMask);
   ListFiles (oldDir, fileMask);
   Thru := TRUE;
 END;

 IF not thru THEN
   Writeln ('PATH NOT FOUND');

 ChDir (oldDir);                     { Restore former directory }
END.
```

End Listing

QC together (minus CV, of course). The main problem with QC is that I think it was shipped much too early as a response to TC. When I received QC as part of my MSC 5.0 update, I had already been using TC for several months. I quickly discovered several things about QC 1.0.

Where TC 1.0 had bugs, QC 1.0 had BUGS. It crashed, it didn't like TSRs, it couldn't handle large or even medium size programs, and it was a memory hog. It apparently interacted closely enough with the hardware to cause damage to hard disk FATs if you had a certain model controller. I was not amused.

The integrated environment provided no control over anything. You were stuck with the medium model (I usually only keep large and small model libraries installed). Coprocessor support wasn't available. Trying to find the limitations of the integrated environment was not easy. I thought that the QC documentation was rather sketchy in this regard.

Unfortunately neither TC nor QC comes with an acceptable editor. Most people consider multiple file capability a minimum requirement. (*The Microsoft ME* editor shipped with the MASM 5.0 also seems to be a klunker.)

At present I use TC for prototyping and MSC 5.0 for production code. For now this seems to be the optimal development environment. Perhaps I will try QC again in the future, but I really don't see anything in particular to recommend it. CV compatibility is nice, but source-level debuggers are only useful in certain situations and CV's size and lack of speed somewhat handicap it for larger programs. I do hope Borland will ship a source level debugger or provide CV support, but I can always switch to MSC 5.0 and use CV.

Michael W. Jeorms
Westmont, Ill.

More On A Standard C . . .

Dear *DDJ*,

God preserve us from language standard committees.

The beauty and power of C has always been that, within very broad limits, it left the decisions up to you, and did not constrict your possibilities within the bounds of what other people thought appropriate. If you substituted your own routine for a standard one, and it worked, all well and good. If it didn't—on your head be it.

In other words, you were supposed to be adult enough to make your own decisions and to accept the responsibility if they went wrong, even if only because of unforeseen, possibly undocumented side-effects.

So, will the ANSI C committee (and any other language standardization committees that may be listening) please note—RESERVED WORDS ARE A BAD THING. Although a necessary evil, they are bad in themselves for psychological reasons, but particularly bad for C, both for the reasons given by Mr. Linke (see *DDJ* "Letters," January 1988) and the fact that existing programs are suddenly liable to stop working—I've had that problem often enough with "improvements" to other languages.

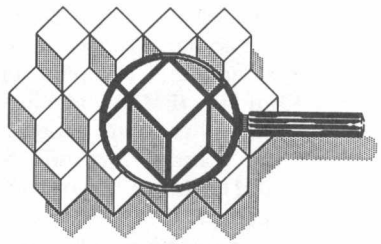
Interested though I am to produce programs which run faster and more efficiently, and which can be ported to other machines more easily, I am most interested in writing programs which work—and everything I have seen so far makes me suspect that that will be more difficult, not less.

C.J. Price
Dortmund 1,
W. Germany

DDJ

We welcome your comments (and suggestions). Mail your letters to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or MCI Mail c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

OF INTEREST



C Products

Software Development Systems has introduced CrossCode C, an optimizing C compiler targeting the Motorola 68000 family. This compiler is designed primarily to address the needs of embedded systems designers. CrossCode C generates ROMable code for all 68000 family members, including the 68020 and 68881 floating point coprocessor.

The compiler comes with a Motorola-style macro assembler, a linker, a librarian, and a C library containing the source to over 47 C functions. Also included is a downloader that communicates with EPROM programmers, emulators, and target hardware. This utility converts user's compiler C code into Motorola S-Records and a variety of other industry standard file formats.

CrossCode C is available for $1,595 under MS-DOS or Xenix, and is available for $4,795 on most Unix-based machines. Reader Service No. 20.
Software Development Systems
3110 Woodcreek Dr.
Downers Grove, IL 60515
312-971-8170

Oregon C++ Software Development System for the Sun-3 is now available from **Oregon Software**. The Oregon C++ compiler directly generates executable code from programs written in C++, Kernighan and Ritchie C, or ANSI C. The system is delivered with a complete ANSI C library and a library compatible with the AT&T stream I/O library for C++. The product includes a source-level debugger and an interface to Sun's dbxtool and is available for $1,900. Network pricing is also available. Reader Service No. 21.
Oregon Software
6915 S.W. Macadam Ave., Ste. 200
Portland, OR 97219-2397
503-245-2202

IBM has released the IBM C/2 Version 1.10, which is a replacement version of an earlier C compiler for the IBM PC and the IBM PS/2 families. It will run on and generate code for IBM personal computing products under DOS and IBM OS/2. It separates code and data, permitting only one copy of the code section to be used for memory in a multi-application environment. The product is available for $560. Reader Service No. 22. Consult your local IBM PC dealer.

McCPrint, a C source code beautifier/reformatter for the Macintosh, has been announced by **MMC AD Systems**. McCPrint is multiwindowed, menu driven, and allows simultaneous manipulation of multiple, large source code files. When beautifying/reformatting C source code, McCPrint provides options that control how braces are placed, how comments are formatted and aligned, how continuation lines are broken and aligned, where spaces are placed, and more.

In addition to standard editing features, McCPrint enables user selection of tab sizes, font type and sizes; automatic highlighting of C keywords, comments, functions, and/or Macintosh toolbox calls; shifting left/right of multiple statements; and assistance in determining the beginning/end of code blocks, strings, comments, quoted strings and constants, and array indexes. The search facility includes support for automatic, multiple file searches, manual and automatic string replacement, and an optional Unix grep capability.

McCPrint generated source code works with most Macintosh and non-Macintosh C compilers. The product sells for $59.95. Reader Service No. 23.
MMC AD Systems
P.O. Box 360845
Milpitas, CA 95035
408-263-0781

Lattice now offers the C++ language for Amiga programmers. Lattice C++ is a preprocessor that translates the C++ language into C source code which is then compiled by the Lattice AmigaDOS C compiler and linked to create an executable program. A driver similar to the LC driver in the Lattice AmigaDOS C compiler is provided to allow translating, compiling, and linking in one step.

The Lattice C++ language allows programmers to declare in-line functions for small frequently-called operations. These in-line functions can replace macros and thereby utilize the same semantics as other functions, yet they retain the code size and run-time efficiency of a macro.

Other features of the Lattice C++ give programmers the ability to declare variables as statements anywhere in the program; it provides a mechanism for expressing commonality among different types by explicitly defining one class to be part of another; it allows a reference, which when initialized, becomes an alternative name for the object it is initialized with; and it provides many other functions designed to improve the C language. The product is priced at $500. Reader Service No. 24.
Lattice Inc.
2500 S. Highland Ave.
Lombard, IL 60148
312-916-1600

Operating System News

AIM Technology unveiled its OS/2-Unix Multitasking Benchmark, a program that measures the performance of the new generation of CISC and RISC workstations operating in a multitasking environment. The program evaluates and compares OS/2 system capabilities with similar Unix systems under a wide variety of application mixes.

The OS/2-Unix Multitasking Benchmark enables the following comparisons: OS/2 system to OS/2 system; OS/2 system to Unix system; and Unix system to Unix system. It allows end users to tailor tests for unique environments and measure the amount of computer processing power that can be applied to a specific mix of application software.

The benchmark uses 31 functional tests to evaluate performance under user-defined application mixes as found in office automation, scientific, and engineering environments. The functional tests can also be used to evaluate the performance of subsystems such as disk drives, math coprocessors, and memory. Users can tailor the program by adding functional tests that are integrated into the benchmark. The program can also be run during regular system operation in order to evaluate actual work environments.

The benchmark is designed to run on all OS/2 and Unix implementations. It is distributed in binary format for OS/2 and 80286/80386 Unix machines and in source code for other Unix machines in a variety of media formats. A basic General License Agreement starts at $3,950. Reader Service No. 25.
AIM Technology
3350 W. Bayshore Rd., Ste. 203
Palo Alto, CA 94303
415-856-8649

LynxOS, from **Lynx Real-Time Systems**, is a real-time operating system fully compatible with Unix. LynxOS was designed to give real-time capabilities to 32-bit microcomputers such as the Motorola 68000 processor family, the Intel 80386, and RISC processors, including SPARC.

LynxOS is fully compatible with Unix, runs existing Unix application software, and provides a familiar software development environment. The program also supports a fully preemptible kernel with minimal interrupt disabling. The highest priority task that is not waiting for resource usage or I/O has the CPU. A task's priority is set by the user and is not changed by LynxOS.

The LynxOS package includes the LynxOS kernel, run-time libraries, device drivers, more than 100 utility programs, and full documentation. Price for a source code license for the initial CPU is $65,000. Reader Service No. 26.
Lynx Real-Time Systems
550 Division St.
Campbell, CA 95008
408-370-2233

Hewlett-Packard has announced HP LaserROM for the HP 9000 Series 800 HP-UX computers, a service that delivers Unix documentation and sup-

port information on CD-ROM. HP-UX adheres to AT&T's Unix System V Interface Definition Issue 2. This service enables the user to electronically search and retrieve information related to a Unix operating system ranging from software design and development manuals to support information. The initial version holds the hardcopy equivalent of more than 10,000 pages in electronic form.

Each significant word on the HP LaserROM disc for HP-UX is indexed. This allows the user to instantly locate specific information within hundreds of megabytes by typing in keywords. Users can specify selected words, phrases, or topics of interest and the system identifies each occurrence of the specified information.

HP LaserROM for the HP 9000 Series HP-UX computer is $1,800 for a 12-month subscription. Reader Service No. 27.
Hewlett-Packard Co.
Customer Information Center
Inquiry Fulfillment Department
19310 Pruneridge Ave.
Cupertino, CA 95014

Tools and Utilities

FaceIt from **Black & White International** is a development tool that creates windows from .dbf and ASCII text files. It designs everything from data entry menus to help windows to complete user interfaces. The program also determines the window dimensions based on the amount of text to be displayed; designs the layout of the windows; arranges window placement on the screen; configures the proportional spacing between items; creates columns and determines the correct number of columns; makes all windows scroll; and allows users to complete menu customization.

FaceIt runs on IBM PCs, IBM PC ATs, IBM PC XTs, IBM PS/2s or true compatibles. The package is priced at $99. Reader Service No. 28.
Black & White International
P.O. Box 21108
New York, NY 10129
212-787-6633

Complete Logic Systems, the developer of a general purpose programming language called Trilogy, has announced the release of the Trilogy Toolbox and Trilogy Linker.

The Trilogy Toolbox includes four modules: the Graphics Module, which allows users to translate, zoom, and rotate graphics objects and supports Hercules, CGA, EGA, and some VGA modes; DOS Module, which contains a number of DOS calls and returns complex information such as directory tree, list of file names, and so on to the corresponding symbolic data structures of Trilogy for further processing by queries and predicates; DOS Util Module, which emulates DOS commands and a variety of utilities; and Pattern-Matching Module, which allows users to perform a syntax analysis of a string.

The Trilogy Linker is a program designed to produce a stand-alone relocatable load file out of Trilogy program modules.

The cost of Trilogy Toolbox is $50 and the cost of the Trilogy Linker is $50 for private use and $200 for software developers. Trilogy with its modules runs on IBM PCs, IBM PC XTs, IBM PC ATs, and compatibles. Reader Service No. 29.
Complete Logic Systems
741 Blueridge Ave.
North Vancouver, BC
Canada V7R 2J5
604-986-3234

Tools & Techniques has released DataPlex, an intelligent front-end software for complete data handling. The core of the product is a uniform data entry engine that accepts data at optimum levels of speed, accuracy, and ease of use independent of destination. DataPlex employs a human interface coupled to AI techniques such as dynamic data dictionaries and pattern recognition to expedite data entry. The product learns validation rules from the inherent nature of the data, and auto-styles field formats accordingly. If necessary, the user can navigate menu options to override with fixed format attributes. Because the connections from DataPlex to different formats have been engineered as read/write two-way data paths, the product serves as a central conversion hub, or dataplexor, for transferring data between applications.

DataPlex was written in Microsoft C and versions are also available for Xenix-286/386 and Unix V-6800. The price for Version 1.0 is $149, and the price for Version 2.0 is $195. Reader Service No. 30.
Tools & Techniques Inc.
1620 W. 12th St.
Austin, TX 78703
512-482-0824
800-444-1945

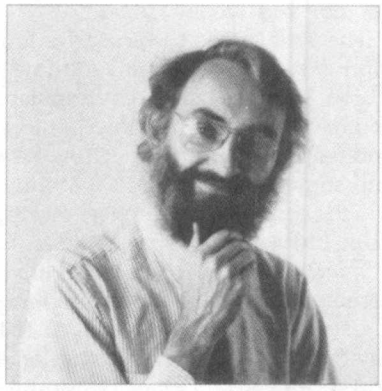
Quilt Computing has announced the release of MacSRMS, a Macintosh version of its Software Revision Management System. MacSRMS is a version control system that can be used by programmers, system developers, project managers, librarians, and system administrators to effectively control the proliferation of the many versions of source associated with programming projects.

MacSRMS stores all versions of source in a single Macintosh library file, providing the user with the ability to retrieve any particular version from the library. MacSRMS supports all programming environments, compilers, and editors that store the source in the form of ASCII characters.

MacSRMS offers a fully functional Macintosh user interface, an integrated editor, and compatibility with MultiFinder and AppleShare. MacSRMS runs on the Mac512KE, the Mac Plus, SE, and II. The price of MacSRMS is $195 when used in a single machine environment. Reader Service No. 32.
Quilt Computing
7048 Stratford Rd.
Woodbury, MN 55125
612-739-4650

DDJ

SWAINE'S FLAMES



Now it can be told. Here are all the snide innuendos, the catty bag emptyings, the bitter recriminations of four years at the pinnacle of personal computer professional software development magazine publication management. Yes, in the honored tradition of former insiders like Donald Regan, Larry Speakes, and Martha Mitchell, here is the Mike Swaine kiss-and-tell flame.

Laird fidgeted nervously as he announced the postponement of the managers' meeting. He claimed that he was doing it because the ad sales managers were out of town, but I knew it was really on the advice of his astrologer. After all, the sales managers were always out of town when there was going to be a managers' meeting.

I admired them for that.

I, of course, was ever the model manager. I must have modeled for about a dozen of those column photos. But besides that, in the four years of my administration as editor-in-chief, circulation quadrupled, ad revenues skyrocketed, and the magazine went from being a struggling nonprofit venture to being the cash cow of the company. I was entirely and personally responsible for this success, to say nothing of the trend in the number of editorial pages delivered per month. I said to say nothing of the trend in the number of editorial pages delivered per month.

Since leaving office, I have had the opportunity to look back over the years at M&T and before that at CW Communications and to reflect on the strengths and weaknesses of the computer press. Sitting here gazing out across the Pacific, I've finally been able to put it all in perspective. I have concluded that most of the leaders in the field would have their vision improved by being outfitted with glass navels. Here's my four-point plan for dealing with the computer press:

- To get noticed by a computer magazine, write a clever press release. Editors learn about new products from press releases, and we throw away the dull ones. The ones that are too technical for us we file. Forever.
- To get a product reviewed, be Microsoft, or, if that's not feasible, create a product with true usefulness or distraction value to magazine editors. Consider desktop publishing. Consider the Radio Shack Model 100.
- To avoid a bad review, don't put an apostrophe in possessive "its." Maybe we can't spell your firm's name, but we know our apostrophes.
- If you get a bad review, write an outraged letter to the editor saying that all those fatal bugs are fixed in the latest release. Ask what good a review is that's already four months out of date on publication. Listen to him squirm.

Enough. I hated the kiss-and-tell genre to begin with, and although I thought the parody was extremely broad, Jon and Peter hated the first two versions of this k&t column. If you hate this one, blame them. Just kidding, heh, heh. The fact is, I am grateful to and fond of the magazines I've worked for, even when I have had to do three versions of a column. But even if I weren't, I'd never write another of these damned k&t columns: the genre doesn't deserve the respect of parody. The only truly serious point in all of the above pseudovituperation is this: when you deal with the computer press, you are still chiefly dealing with amateurs. Don't trust us blindly.

Now if nobody minds, I'll get back to the true mission of this column: biting the hand of the computer industry. Enough of this kiss-and-tell stuff already.

I predict that large chunks of Teflon will chip off John Sculley as a result of an anonymous kiss-and-tell book. Curious technical passages will befuddle reviewers until it is learned that the book is actually a draft of the manual for the NeXT computer.

I predict that Apple will not introduce card racks into the next version of HyperCard and that ex-Xerox PARCer Paul Heckel, author of ZoomRacks and *The Art of Friendly Software Design*, recently granted a patent for the card-rack metaphor, will not sue Apple, although he will continue to end every sentence with "if you know what I mean." I predict that Heckel will write a book about PARC, if you know what I mean. The book will make heavy use of analogies to the film industry and nobody will know what it means.

I predict that, again this year, no kiss-and-tell books will be coming out of IBM.

Michael Swaine

Michael Swaine
editor-at-large

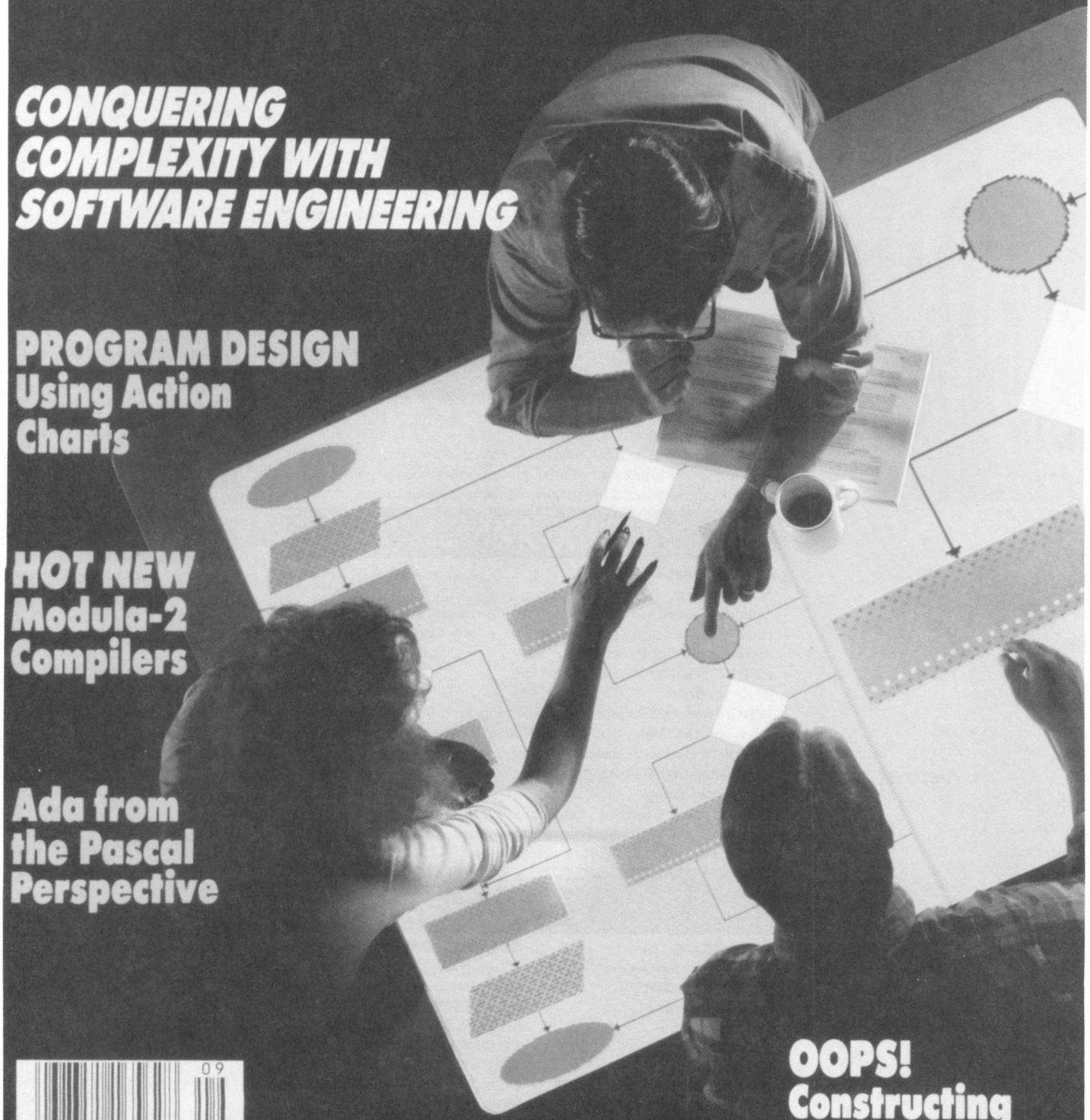
#143 SEPTEMBER

2.95 (3.95 CANADA)

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER



CONQUERING COMPLEXITY WITH SOFTWARE ENGINEERING

PROGRAM DESIGN Using Action Charts

HOT NEW Modula-2 Compilers

Ada from the Pascal Perspective

OOPS! Constructing Object Classes

09
0 38351 16562 8

ARTICLES

Program Design ▶ **Using Action Charts** 18
by Martin Stitt
In the time honored less-is-more style you've come to expect from *DDJ*, Martin shows you how to use keyboard macros with a word processor to create action charts from your source listings.

Ada ▶ **Ada for Pascal Programmers** 34
by Kim King
Getting acquainted with Ada needn't be a stressful experience especially if you're already comfortable with Pascal.

Object-Oriented Design ▶ **Object-Oriented Dimensional Units** 52
by John A. Grosberg
Tired of a classless society? John presents an object-oriented class structure using dimensional units.

Arguments and Automatic Variables in Assembly Language 82
by Raymond Moon
One problem with assemblers is that they can't reference passed arguments and local variables. Ray presents some techniques for getting around this limitation.

HyperTalk ▶ **XCMD & XFCN: Hypercard's Software Slots** 90
by Stan Krute
Using the software hooks which Apple has so thoughtfully provided, Stan explains how you can extend HyperTalk to include almost any function or procedure you might need.

REVIEWS

Modula-2 ▶ **Examining Room** 66
coordinated by Ron Copeland
This month our senior technical editor, Kent Porter, puts a handful of Modula-2 compilers through their paces and emerges from the process as a born again Modula-2 programmer.

COLUMNS

C PROGRAMMING 116
by Al Stevens
Al begins a multi-issue project which develops a communications program and talks about the agony and the ecstasy of incremental implementation.

STRUCTURED PROGRAMMING 126
by Kent Porter
Kent's Modula-2 madness continues as he follows the development of the language as it evolved (initially) from Wirth's dissatisfaction with Pascal's limitations.

Neural Nets ▶ **PROGRAMMING PARADIGMS** 148
by Michael Swaine
This month Michael examines Dennis Reinhardt's Nerl Netwurkz, a tool that lets you build a neural net by constructing, then connecting individual neurons.

FORUM

EDITORIAL 6
by Jonathan Erickson

RUNNING LIGHT 8
by Ron Copeland & Michael Swaine

LETTERS 12
by you

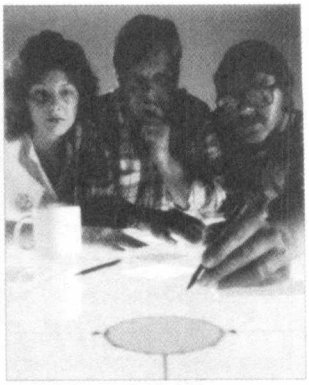
SWAINE'S FLAMES 160
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 145
where to go for more information on products

OF INTEREST 156
brief product descriptions

PROGRAMMER'S MARKETPLACE 157
classified ads



Next Issue

DDJ's October issue examines extensible languages like Lisp and Forth. On other fronts, we'll take a look at the 80386's protected mode (and how to write 32-bit software for it), share an assembly-language cross-cross referencing tool, and evaluate Turbo Prolog 2.0. All this in addition to our usual collection of columns and columnists.

EDITORIAL

If you're one of the few dozen or so readers who have recently been tuning in to the late night chitchat on *DDJ*'s CompuServe Forum, you might have been led to believe that *DDJ* is in for some change. Over the past few weeks, you see, much of the speculation on the part of Forum participants has revolved around the present and future focus of the magazine and the direction in which *DDJ* is heading (editorially) in the coming months. And with all those strange names on the masthead of late, it sure looks like change is in the air. Obviously, you might go on to think, there's trouble brewing.

Some concern is natural, and believe it or not, we—new and old editors alike—are kind of happy to find out that *DDJ* readers care enough about the magazine to raise a stink when they feel their magazine is being threatened. We appreciate hearing from you when you think we're doing a bad job as well as when we're doing a good one.

This question keeps coming up so it must be important: Is *DDJ* going to cut back on the size or number of program listings? Or maybe even stop printing them altogether? Relax. It's not going to happen. The answer is, and will continue to be, *no*.

Each month we'll publish program code, and we'll continue to make the source files available either directly (through M&T Books, for a nominal charge) or for free via electronic online systems such as the *DDJ* Forum on CompuServe. This is a continuing commitment. We recognize that listings are important. (It's my view that it's the text that tells you what the author has in mind, but it's the code that tells you what the author is really saying.)

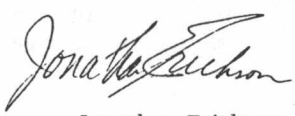
Another question that keeps cropping up concerns some of the changes we've made in our columns (or columnists). *DDJ* is currently running three monthly columns: one on C programming, one on structured programming, and the enduring Mike Swaine's column on programming paradigms. These are our core columns.

The reason we felt we needed to run fewer columns is that we wanted to be able to run more features. Running a handful of columns each month (complete with all that code) takes away from our ability to run a balanced lineup of articles. Both provide information. Both are important. We just thought we needed to provide a better balance.

Perhaps a more critical concern voiced by some readers is the perception that *DDJ* may be moving mainstream, in pursuit of the mythical PC volume buyer. Not so. We know our readers. We know that you are experienced programmers who want to be challenged, not coddled.

What you can expect to find each month in *DDJ* is a monthly theme that is supported by at least two articles. These articles will be followed by an additional two or three articles (which may or may not be related to the theme) that typically discuss programming techniques. For the most part, these will be articles that provide practical solutions to day-to-day programming problems (usually code-intensive). However, we'll also try to cover a topic that may not be a solution to a problem today but that may be a concern to programmers x number of years from now.

So, the good news is although the medium may change, the message remains the same — *DDJ* will continue to provide its readers with the best technical information available about computers, programming, and that intriguing pseudo-space where they come together.



— Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Managing Editor *Monica E. Berg*
Senior Technical Editor *Kent Porter*
Associate Editor *Ron Copeland*
Editorial Assistant *Kathleen Evans Ralston*
Technical Editor *Richard Relph*
Contributing Editors *Al Stevens, Martin Tracy, David Betz*
Copy Editors *Kevin Shafer, Rhoda Simmons*
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Lynn Sanford*
Typographers *Lorraine Buckland, Margaret Anderson*
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Asst. Circulation Manager *Andrea Weingart*
Newsstand Manager *Sarah Frisbie*
Fulfillment Coordinator *Francesca Martin*

Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Marketing Assistant *Sara Noah Ruddy*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, except semimonthly in June and October by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the associate editor.

***DDJ* on CompuServe:** Type GO DDJ

Subscription: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds with additional postage. Add $13 per year for surface mail to all countries or add $33 per year for airmail to Canada and Mexico; add $32 per year for airlift/surface to all other countries.

Customer Service: For subscription orders and changes of address call toll-free 800-354-8400 or write: *Dr. Dobb's Journal* subscription Dept., P.O. Box 3771, Escondido, CA 92025. For all other subscriber inquiries call 800-321-3333 (in California 800-331-4164, outside the U.S. 619-485-9623). For book/software orders call 800-533-4372 (in California 800-356-2002).

Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3771, Escondido, CA 92025. **ISSN 0888-3076**

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

ABC

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

So what does software engineering mean? Well, the answer you get, as is often the CASE, depends upon who is answering the question. For some readers, the answer is likely to be putting a bridle on innovation. What thwarts the creative process aborts the created product. For others it's just as likely to mean a way to produce better products, in less time, with less waste, and have more fun in the process.

Definitions of software engineering tend to lead to arguments. So for the sake of argument let's say software engineering refers to a technological and technology management discipline concerned with producing and maintaining software products on time and within cost estimates. It doesn't necessarily refer to team programming, but because it is impossible to guarantee that a software product will be maintained by its original author, it does necessarily preclude the kind of clever code that only the original author can understand. Every software product, in this view, is in fact a multiprogrammer project and should be approached as such. Communication is a central issue.

Traditionally, approaches to software engineering have involved most of the following:

- Techniques for time and cost estimation
- Project planning for the life cycle of the software product
- Verification, validation, and other assessment techniques
- Concern for organizational structure
- Formal specification languages
- Particular techniques of software design and implementation
- Shared tools.

Fortunately, over the past few years, several companies have developed a collection of specialized methods and tools which address the complex problems of writing and maintaining large programs. These powerful tools are now becoming more readily available on micro platforms.

The nature of software development is such that decisions of sweeping magnitude can be made and implemented in seconds, and, in fact, this goes on routinely. It is simply impossible for a programmer to break off the programming process and explain the significance of a change he has made every time he does something that could affect a distant portion of the program. One of the ways software engineering attempts to address this problem is to make it impossible for the programmer to affect a distant portion of the program by creating an organizational structure that matches a natural decomposition of the problem. Each programmer (or small group of programmers) works, not on the whole problem, but on a component.

The results of the software engineering efforts, when they work, are predictability and maintainability. The perception among many programmers, including, we believe, a high proportion of *DDJ* readers, is that this makes programming boring and unimaginative drudge work. This perception must be at least partially true, and the way out of the dilemma seems to be something like CASE: computer-aided software engineering. Rather than force programmers to do boring and restrictive things, the CASE approach automates the boring activities. The programmer's options are still restricted, but he doesn't notice because he is spending his time in areas where he is not restricted. CASE is, to the programmer, a friendly environment and a set of tools for software development. At least that's the theory.

Ron Copeland
&
Michael Swaine

ARCHIVES

Ten Years ago in *DDJ*

"The techniques of engineering programs are being developed. One of them is modular programming which, when applied to [compiler] translators, depends upon the organization of the translator. . . . The time should come when translators are well-engineered as a rule, not as an exception."—*William M. McKeeman, "Programming Language Translation Techniques"* DDJ, *September 1978.*

In Defense of Ada

"Considering Pascal's wide acceptance, and Ada's similarity to Pascal (which is not a coincidence), I am at a loss to explain the evident lack of support for the new language. . . . The language has its faults. Operator overloading certainly won't make a program more readable or maintainable, the I/O stinks, and most people think that Ada tried to bite off more than it could chew. Nevertheless, one fact remains: you can bet that good Ada programmers will soon be in demand."—*"Letters,"* DDJ, *November 1983.*

The Price Sounds Right

"Shugart now has several of their new, fixed-spindle, 30-Mbyte drives running and roaming. The rumorists couldn't agree, however, on whether Shugart marketing had settled on $1,500/unit or $750/unit. Oh well, for a 30-Mbyte hard disc, we might be willin' to cough up a kilobuck, more or less.—*"Silicon Valley Rumors at the Homebrew Computer Club,"* DDJ, *August 1977.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Real-Time Reaction

Dear *DDJ*,

I was delighted to see an issue devoted to real-time programming (June issue). As a person who has followed and worked with real-time Unix on PCs for many years, I recognize it as being a vehicle for fixing many of the serious complaints people have with Unix.

What the articles failed to convey are some of the dramatic effects real time has by turbo-charging some commonly used Unix software. Too *often people* have just focused on communication servers or special time dependent access to hardware. It is true that real time is important here, and some of the most important effects of real time are dramatically realized in these areas, but performance improvements can occur in many other areas.

For example, the server technology improvements due to real-time scheduling, which Dan Hildebrand used to make his communications servers work reliably, can also provide dramatic improvements in X-windows as well as other windowing software. Problems which are solved are Mouse tracking on a loaded system, and a signaling process doing intensive I/O to the screen. Real time can give a totally different feel to the responsiveness of X-windows.

Real-time graphics systems, such as those used to trace the path of a football or weather pattern on your TV have been employing real-time Unix for many years.

Networking servers, often a bottleneck on distributed systems—particularly on the remote file system (RFS) implementations—can be significantly improved.

In addition, the database world of on-line transaction processing (OLTP) benefits from the real-time technology. A base-line factor of 2x improvement in throughput is achieved with asynchronous disk I/O. Asynchronous I/O allows processing at a user level to continue while a read or write system call is being processed. Direct disk I/O, allowing multi-block transfers to occur to the Unix file system, can produce a factor of 10x improvement in some cases.

In the factory automation process control world, where interrupt latencies and rapid context switching are important for basic real-time control of machinery and monitoring systems, a revolution in software is occurring. The availability of cheap high performance CPUs have meant that real-time Unix systems with general-purpose software can be used, whereas previously only dedicated controllers with special purpose software were required.

Many companies are moving quickly to get these real-time features for some of the basic features described above. Venix/386 and Venix/286, AT&T certified ports of Unix System V to the PC AT and compatibles, contain all these features of real time and many more. VenturCom has ported the real-time enhancements of Venix to Interactive System's 386/IX and is now working with DEC to incorporate them into Ultrix. There is more to come in the future.

It is nice to see real-time technology take its place among standard desirable features of an operating system. In the past, it has been relegated to only process control applications or to scientific experiments. It works there as well, but its features offer much more than the name suggests.

William P. Spencer, PhD
Cambridge, Mass.

To his credit, Melvynn did not let his utter lack of wit, charm, or rudimentary hygiene deter him from a lucrative job "in computers."

4.0 Versus Two Compilers

Dear *DDJ*,

When I read Kent Porter's comparison of Turbo Pascal, Version 4.0 and Microsoft Pascal, Version 4.0 (June issue), it occurred to me that the difference in execution times for some floating point benchmarks stems primarily from the differences in the numerical format used by the two compilers. One may observe the same effect in the reduction in number-crunching speed between QuickBasic 3.0 and 4.0 when used without the assistance of an FPU.

Mr. Porter mentions that the last set of benchmarks were run without an FPU. Thus, the Turbo Pascal version would use the old six byte "real" type, which is obligatory on machines lacking an FPU; whereas, the Microsoft Pascal would be using IEEE floating point numbers and FPU emulation in software. I would guess that any additional optimization the Microsoft compiler might do would tend to pale to insignificance when compared to the substantial loss of speed due to the FPU emulation. The same effect is somewhat noticeable, though less pronounced, in the "fib" benchmark. If I'm correct, the differences should largely disappear on a machine equipped with an FPU. With the fact that only a small percentage of micros have an FPU, one can reasonably question Microsoft's decision to go to exclusive use of the IEEE floating point number in their latest series of compilers.

As to the size of the executables produced, Microsoft's linker still (unless recently changed) appears to include any and all runtime library functions in the executable, regardless of whether or not the application needs them. I expect that Microsoft collectively feels that with the increasing memory sizes available in micros, the size of executable files is irrelevant. I disagree and live in hope that they will soon bring out a "smarter" linker. I also applaud Borland's linker's intelligence and hope that they will make it available in a stand-alone form in the near future.

Dan Dixon
Woodward, Okla.

Pro-Screen Similarities to PC/Forms, Version 1.21b

Dear *DDJ*,

I just read Kent Porter's review of PC/Forms, Version 1.21b in the May issue and was struck by the resemblance to Zortech's Pro-Screen. I purchased the Turbo C version from Zortech Inc. for $50.

Although I have not had the time to use Pro-Screen enough to be thoroughly familiar with it, there are a few quirks which may or may not prove to be serious. First, it is produced by a British company so all the financial functions are in pounds and pence (at least the documentation is). Second, while sold as a stand-alone library, it is designed primarily to work with Zortech's window library. And, third, the emphasis is on business functions: the Zortech B-tree/ISAM library is touted. So, Zortech's Pro-Screen is not quite the general purpose library if you believe the documentation; so far, however, it is doing what I bought it to do.

Richard B. Shepard, PhD
Palatka, Fla.

Clarification

In the August issue on "Speed Trials: Five Cs Compared," Version 6.0 *and* Version 6.5 of Watcom's C compiler were benchmarked. *While the text* of the article accurately pointed out that Watcom C 6.5 outperformed all other compilers evaluated — running the Dhrystone at 8.8 seconds — Table 1 made reference to only Version 6.0 (which performed the same test in 10.6 seconds). Although both figures are correct, both versions should have been included in Table 1.

On page 24 of the same article, the captions for Examples 2 snd 3 were reversed. They should read: "Example 2: Test suite results for Borland's Turbo C, Version 1.5" (no errors detected) and "Example 3: Test suite results for *Watcom C, Version* 6.0" (failed). We apologize for this discrepancy. — ed

DDJ

USING *ACTION CHARTS* FOR SOFTWARE DEVELOPMENT

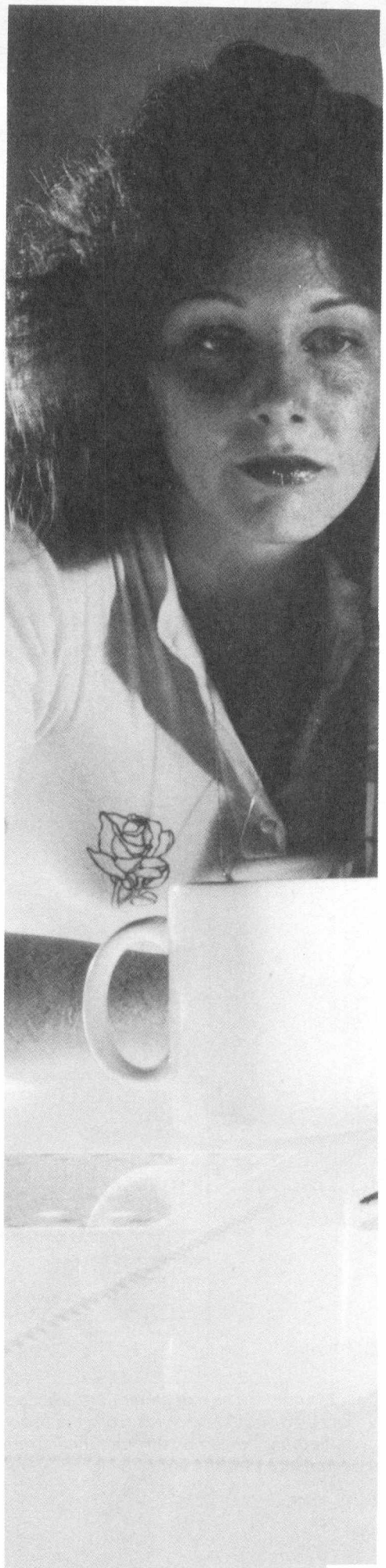
by Martin Stitt

In my work as a software engineer, I have experimented with a variety of diagramming methods for software design and documentation and have taken a great interest in software engineering tools in general. What I have discovered is that, although software engineering tools are becoming more available in the PC world, their high price still keeps them out of the reach of many developers. In this article, I will show how you can implement a technique known as action-chart diagramming — that is, the use of brackets to illustrate a program's structure graphically — with nothing more than an editor and a set of keyboard macros. Although the tool I describe is not as comprehensive and sophisticated as a full-scale commercial package, it is quite capable and is essentially free.

A primary objective in applying software engineering tools to software design is to offload to the computer as much of the clerical and mechanical details inherent in program development as possible. The programmer's talents can then focus on actual logic design and verification. A principal technique in the implementation of a CASE toolset is diagramming, and although many different diagramming methods exist, each with their own strengths and weaknesses, I shall limit my discussion here to action charting. This method is well suited to the development of procedural logic in almost any language and is easy to draw on a computer.

You may already be familiar with formatting utilities that produce a printed listing of a source code module annotated with action-chart style brackets. One example is Source Print, produced by Alderbaran Laboratories. By illustrating the nesting and grouping of program statements graphically, these utilities can provide a clearer picture of what's going on in a program. Although I certainly do not wish to deny the usefulness of these aids, applying action charts after you've created a source module is like buying a road map only after you've got lost. It is possible to develop program logic using such brackets from the very beginning and not have

Martin Stitt is a software engineer for The Software Link Inc. He is a member of the design team for PC-MOS/386 and is currently writing a book on it (to be published later this year). He can be reached at 705 Chestnut St., Eugene, OR 97404.



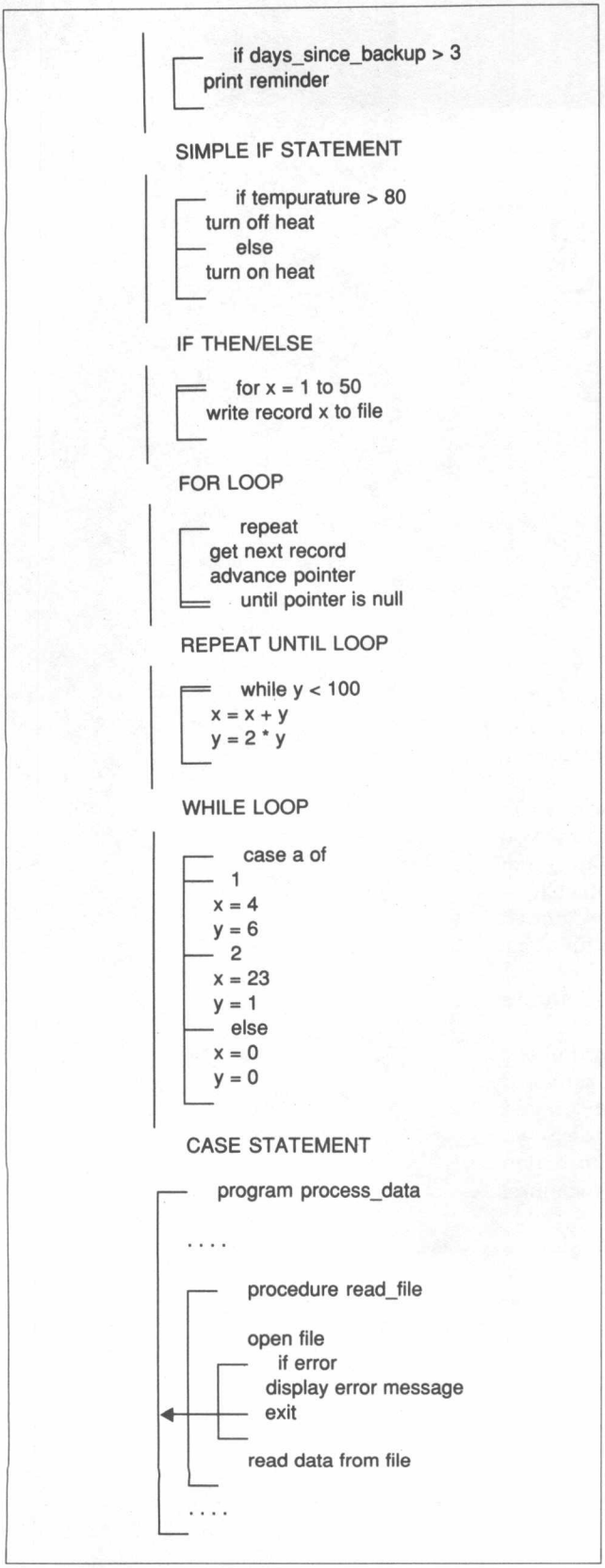


Figure 1: *The exit statement indicates that an immediate return should be made from the procedure read_file. Running the arrow through the leftmost bracket would indicate that the entire program should be halted.*

(continued from page 18)

to waste time being "lost." And the benefits can be realized throughout all phases of the project's life cycle.

To demomstrate how you can turn an editor into an action-chart development workstation, I will describe how to set up macros in WordPerfect; you can use almost any full-screen editor or word-processing program, however, if your editor has a built-in macro facility, so much the better. If not, a ProKey-type utility should provide the needed capabilities. To produce the brackets of an action-chart within an editor, you need to use certain block-graphics characters that are part of IBM's extended

Action charts and the implementation of common-keystroke macros within an editor are a way of making a certain class of software engineering tools immediately available to anyone.

character set (see the sidebar "How The Macros Work," page 22). These characters are available even on a simple monochrome display and on many popular printers as well, so you should not need special hardware. One exception I've found is that some printers will not print double-lined block characters properly (an Okidata 293 will, but the smaller Okidata 193 prints them as single-lined characters).

A little-known fact about the PC AT computer is that by holding down the Alt key and entering a decimal number from 0 to 255 on the numeric keypad, you can enter any character code, including control codes, normal alphanumeric characters, and the extended character set. The extended set ranges in value from 128 to 255 and includes foreign-language characters, block graphics, and mathematical symbols. The only exception I know of is within the PC-MOS/386 multiuser/multitasking operating system, in which you use such keystrokes to select the task currently visible at your console. In this environment, you can key an Alt-999 to disable this special use of Alt and numeric key pad keystrokes.

In addition to being well suited for editing on a PC, action charts have some distinct advantages in their own right. The illustration in Figure 1, page 20, shows how closely the graphic constructs of action-charts parallel the logical constructs of a typical high-level language. By using brackets you can build a consistent, well-organized structure. Unlike old-fashioned flowcharts, which allow the design of scattered spaghetti-like logic, these brackets implicitly support the one-entry/one-exit rule of structured programming — a design method that actually makes it difficult to put together badly structured code. About the only way to develop a poorly organized program flow is to abuse the *EXIT* mechanism or to concoct some other use of *GOTO*s.

How The Macros Work

Alt-nnn	Enter the decimal value 'nnn' on the numeric keypad while holding down the Alt key
Alt-c	Press an Alt character combination (e.g. Alt-L)
Shft-F2	Hold down the shift key and press F2
F2	Press F2 alone
up	The up arrow key
down	The down arrow key
left	The left arrow key
right	The right arrow key
space	The space bar
Hrt	The enter key
BS	The backspace key
Home	The Home key
End	The end key

Listed above is the notation that will be used to describe the keystrokes used in making the action chart macros.

┌ 218	│ 179	└ 192	– 196
═ 205	├ 195	╔ 213	╘ 212

Shown above are the block graphics characters used to produce action charts and their associated decimal values. When this number is entered on the numeric keypad with the Alt key held down, the corresponding graphic symbol will appear.

The macro Alt-L is used to make a bracket. Before invoking this macro, place the cursor where you want *the top left corner* of the bracket to be. Two more lines must already exist in the file below your starting point. The keystroke sequence for this macro is:

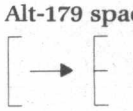
Alt-218 Alt-196 space left left left down Alt-179 space left left down Alt-192 Alt-196 space left up up right

The macro Alt-P enlarges a bracket by one line. The new line is added below the current one; below the line the cursor is currently in. The keystroke sequence for this macro is:

End Hrt Alt-179 space

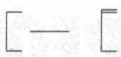
The macro Alt-O fills a gap in a bracket with a vertical bar. When enlarging a set of brackets, Alt-P is used to start the new line and insert the vertical bar for the leftmost bracket. Then Alt-O is used to add the vertical bars for all remaining brackets. The keystroke sequence for this macro is:

Alt-179 space



The macro Alt-K converts the vertical bar of a bracket to a CD symbol for the ELSE portion of an IF THEN/ELSE statement. Start with the cursor located two columns to the right of the point where you want the CD symbol to appear. The keystroke sequence for this macro is:

BS BS Alt-196 space



The macro Alt-G converts a single line top corner into a double one. Start with the cursor located three columns to the right of the top left corner of the bracket you wish to convert. This is where the cursor will be immediately after using Alt-L to make a bracket. The keystroke sequence for this macro is:

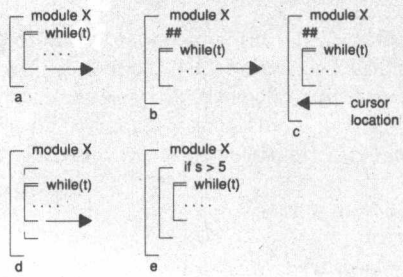
BS BS BS Alt-213 Alt-205 space

The macro Alt-B converts a single line bottom corner into a double one. Start with the cursor located three columns to the right of the bottom left corner of the bracket you wish to convert. The keystroke sequence for this macro is:

BS BS BS Alt-212 Alt-205 space

The macro Alt-R copies the line the cursor is on to a new line below. This is especially useful for enlarging a set of nested brackets.

Home Home Home left Alt-F4 down Cntrl-F3 2 Cntrl-F3 5 up End



Macros Alt-I and Alt-U work together to accomplish the task of inserting a new bracket to the left of one or more levels of pre-existing rackets. In the example above, after the bracket for the while loop was created it was discovered that another test must be added. In order to subordinate the while bracket under a new bracket we must first do a little manual setup. A new line must be inserted just above the bracket to be subordinated (see the Alt-P and Alt-O macros above) and two ## characters must be entered as in (b). Next, a new line must be added just below the bracket to be subordinated. Then, with the cursor located as shown in (c), invoke the Alt-I macro. This will create the top and bottom corners of the new bracket as shown in (d). Now, use the Alt-U to fill in the vertical bars between the new corners.

The keystroke sequence for the Alt-I macro is:

Alt-192 lt-196 space Shft-F2 ## F2 BS BS Alt-218 Alt-196 space left left left down

The keystroke sequence for the Alt-U macro is:

Alt 179 space left left down

(continued from page 22)

As a means of representing procedural logic, diagramming methods offer intuitive clarity by illustrating process flow pictorially. Aside from picking up a few new keyboarding habits to work with the macros, the learning curve for action-chart development is fairly insignificant. And an additional benefit is that this clarity can result in better communication of your ideas and plans to others. When you need to prepare a design specification, whether it be solely for your own use, to distribute to others in your programming team, or to review with clients prior to coding their application, drafting pseudocode in action-chart format is an excellent form of documentation. Such a picture can be worth a thousand words, especially when your audience is not technically oriented.

Action-charts can also be valuable in other phases of a program's life cycle. Their value as a form of documentation should be obvious, and although updating a chart every time you modify source code is just one more thing to attend to, don't neglect it. The effort involved can more than pay for itself down the road, and it's much more practical than updating a hand-drawn chart. It is also possible to develop a program directly in an action-chart document and then filter off the brackets to obtain a compiler-ready source file. This way, the documentation *is* the source file, so the update problem does not exist.

Another fairly common situation in software maintenance is having to modify someone else's code. If you need a detailed understanding of a program you've never seen before, reverse charting it can greatly increase your chances of adding more new features than problems.

Decomposing a Program With Action Charts

When you've got a hot new idea for the next best-selling software package, you will probably want to start writing code immediately. Although this may feel productive, it can often end up not being so. Becoming immersed in low-level details with little or no preplanning can mean creating procedures and data structures that with a broader perspective you would have written differently.

I've found that even when it seems as though a certain program should take only about three days to write if I just jumped in and start coding, it's much better to spend the first day or two developing a complete plan and running tests on high-level models. It can feel strange to not see any tangible results for a couple of days (for example, source code that compiles and runs), but the usual result is that it takes me only a few hours to write up the code once I've developed a thorough plan. The debugging time is also significantly less than if I had neglected the higher-level planning, and the code produced tends to be much cleaner, more maintainable, and more portable.

Once you become used to such a practice and start realizing its benefits, your definition of "tangible results"

ACTION CHARTS

will change. Seeing a well-constructed plan come together and knowing how it can improve your overall productivity can be just as gratifying as seeing code actually execute right away.

In certain situations, however, you may find it helpful or even necessary actually to develop low-level procedures early on. A good example is a procedure that is dependent on external events—a low-level I/O driver for an operating-system or process-control application, for instance. You may have to write an initial version of such a driver to determine just what degree of control is possible and how best to implement the system. It has been said that the best approach to design is not just from the top down or bottom up but a balanced mixture of the two. Too much of either can mean a narrow perspective that will cost you in the long run.

The exact steps involved in this decomposition process will vary depending on the complexity of the project and the programmer's experience and preferred methods of analysis. Still, the two key steps are a thorough round of planning before you write code, and a disciplined ap-

```
program actfiltr

PURPOSE: Read each line of an action chart file and
   filter out the bracket characters. Format the remaining
   text with regards to the left margin and indentation.

DATA DEFINITIONS:

....

SUBROUTINES

   procedure advance index

   PURPOSE: advance index past all characters in a
      specified set

   DATA DEFINITIONS:
   ....

   SUBPROCEDURE LOGIC

      if ....
      ....
      ....
      else
      ....

      ....

MAIN PROGRAM LOGIC

init vars
validate parameters
open source and target files
   while not end of source
   read a line from the source file
   call advance index
   write line to target file

close files
```

Figure 2: *A program's documentation and psuedo code are enclosed within a parent bracket. Subprocedures are setup in a similar way and are nested one level deep under the parent bracket.*

proach to breaking down the program's logic. This is where action-charts are especially appropriate.

Action charts are suitable for developing all phases of a program, from high-level pseudocode down to detailed low-level logic. As mentioned earlier, you can even work down to the source-code level in action-chart format and then use a utility program to filter off the brackets, resulting in ready-to-compile code. If you develop your practice accordingly, you could make all updates to the chart version of your program, filter again, and then recompile—never actually editing the source code file itself. The filter program then becomes in effect a preprocessor which can be run from a batch file or *make* facility, just as any other translation process. In this way, your chart is always up to date and you're working with the code at a much higher level.

Among the steps involved in designing a program are:

1. Gather facts about required output and input data and general program function.
2. Set up data analysis charts and models and develop ideas about the processes to be used.
3. Consider performance requirements, storage needs, *user interface*, and so on.
4. Use screen-painting and modeling tools to set up and exercise a prototype of the user interface.
5. Organize and refine all facts and ideas gathered in the previous steps in order to develop high-level project documentation and an initial pseudocode version of the program's logic.
6. Review your initial pseudocode. Does it fulfill your needs? Is it efficient? What processes are common and could be developed as subprocedures? How can modules be organized to allow for program growth?
7. Make successive passes through the pseudocode, refining it until a level of detail is reached at which an implementation in the target language is evident.

My preference is to use an outline processor (I use MaxThink) to gather and organize the facts, ideas, and considerations and build a data dictionary. This allows me to enter ideas as they come to me without worrying about putting them in any particular order. Once I'm ready, I organize this raw data to develop three types of documentation: a sequential logic flow description, internal program documentation, and public documentation (for example, for the users' manual and promotional material).

Throughout the decomposition process, periodic reviews are important. Going from an overview level to a more detailed one involves a cycle of reviewing your logic at its current level, picking a section and expanding it to greater detail, then reviewing again, and so forth. At each review, run through a checklist of questions: Is the chosen algorithm still valid? Is the logic true to the algorithm? Are the data structures efficient? What operations are common and suitable as subprocedures? What use can I make of procedures presently in my library? Which new procedures should be added to my general library and which are specific to this project? Am I creating a structure that will permit growth?

Action charts are excellent when working with well-

```
program actfiltr

PURPOSE: Read each line of an action chart file and filter out the bracket
characters. Format the remaining text with regards to the left margin and
indentation.

DATA DEFINITIONS:

{ In actual practice, should develop a data design to significant detail before }
{ program logic is planned. Neglected here for sake of brevity. }

SUBROUTINES

    procedure advance index

    PURPOSE: advance index past all characters in a specified set

    pass in a string, a starting value and a test set
    pass back an updated index

        while true
            if index > string length
            exit    { found end of string }

            if the indexed character is not in the test set
            exit    { found start of string's next portion }

        increment index


MAIN PROGRAM LOGIC

init vars
validate parameters
open source and destination files
{ manage file errors ** }

    while not eof(source)
    read a source file line into work str
    index = 1
    indent adjust = 0
    call proc to advance index past all characters in pass set
        if string not null
            if work str[index] in top set + bot set + else set
                if work str[index] in top set
                adjust = 1
                else
                    if work str[index] in bot set
                    decrement indent level
                        else    { work str[index] must be in else set }
                    decrement indent level
                    adjust = 1

            call proc to advance index past all characters in all sets


    { The index now points to the position in work str which is past all the }
    { action chart characters. Now it is time to apply the desired margin and }
    { indentation settings and write the resulting string to the destination file. }

        if string not null
        form the indent string based on:
                desired spaces and/or tabs for left margin
                desired number of spaces per level * current indent level
        concatonate the indent string with the text portion of work str
        write the completed string to the file
        else
            if flag set to pass null lines on to the output file
            write a CRLF to the output file

    indent level = indent level + adjust
close files

{ manage file errors ** }
```

Figure 3: *This is the chart from Figure 1 after it has been expanded. The program (see Listing One) being developed here will act as a filter for action-chart documents.*

defined logic, but when your ideas have yet to be solidified, there's nothing wrong with reverting to a less restrictive method. I often make a freehand sketch in an informal flowchart format, and then when the logic is defined, I edit the flowchart sketch directly into an on-line action chart. I'll cover the process of translating flowcharts to action charts in greater detail a little later. As with many parts of this technique, the best way is to experiment and find the methods that suit you best.

A good place to start is to set up the overall layout of the chart by making a bracket that is labeled with the name of your program. (Refer to sidebar "How the Macros Work" details on how to use the macros to make a bracket.) At the top of this first bracket, place a block of comments that describe the purpose of the program and significant details about the implementation. Following this should be a dictionary of global variables and data structures—which may include actual declarative statements in the format of your target language. See Figure 2, page 26.

By this time, you should have a good idea of what subprocedures you'll be using, although further additions will usually occur as you go through the decomposition process. You may choose to develop the main procedure first, leaving the subprocedures as stubs for now—or you may want to develop the subprocedures first. Either way, make a bracket for each subprocedure within the first bracket and include a statement of purpose and a data header. This header should include what data will be passed in and out, a dictionary of local variables, what global data is referenced and changed, what other procedures are called from this one, and which points in the program call this procedure.

When using precompiled procedures from a library, you may want to keep a file of action-chart excerpts for each procedure. You can then cut and paste these into your working chart file. These excerpts could include just an overview of the procedure or the entire logic flow—as you choose.

Converting a Chart to Source Code

Although action diagrams are useful during the development, testing, and maintenance phases of a project, you cannot feed such documents directly into a compiler. To move from the planning stage into actual code production, you must produce what I'll call a final source code file. This file contains a complete source code version of your program that is ready to be translated into an executable form via an assembler or compiler.

You can use three basic approaches to reach this stage:

1. Develop the program's logic to a moderate level of detail in action-chart format using pseudocode statements that are detailed enough to be clear but not in the exact syntax of the target language. Then use the action-chart document as a guide while manually typing the actual source code.

2. Create an action-chart document that contains complete source code statements, and use a simple filter program to remove the brackets. This will either result directly in a final source code file or in a file that requires only a small amount of manual touch-up to reach the final source stage.
3. Use a more sophisticated filter utility that can interpret the action-chart document and automatically convert it to a final source format, as shown in Figure 3, page 27. This will relieve you from being concerned with an entire layer of detail, such as where to put semicolons and *begin/end*s or curly braces.

Obviously, the third approach is the best one; however, a filter of that level of complexity is beyond the scope of this article. The filter in Listing One, page 92 (written in

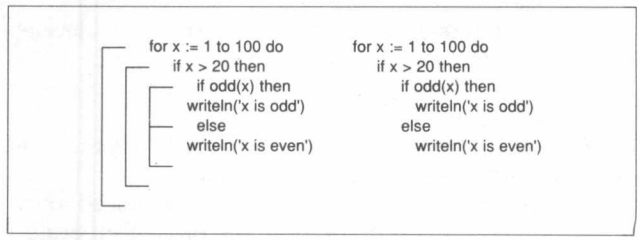


Figure 4: *Through the use of a filter program, the graphics characters of an action chart can be removed and text formatting rules applied to control the left margin and indentation levels.*

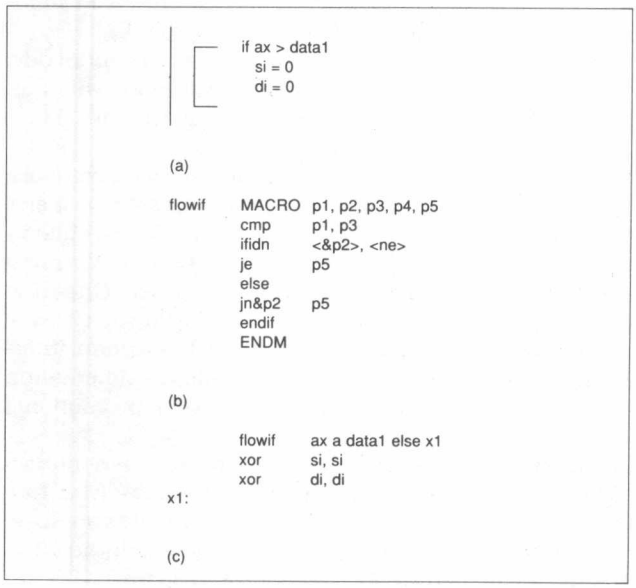


Figure 5: *A statement of the type "if ax > data1" is a positive logic expression; the resulting assembler code would be of the negative logic form. In (a), it needs to be stated that if ax is NOT greater than data1, then the next two instructions must be skipped. In (b), a macro is used to keep the association between the program's action chart and its source code more direct. In (c), the assembler code would read "flow through if ax is above data1, else skip to the label x1."*

Turbo Pascal 3.0), serves as a useful tool when you take the second approach. First I'll examine how this simple filter works and then discuss what could be done to make it more sophisticated.

Figure 2 represents the general logic of a file-processing program. Each line is read in from the input file into a line buffer where it is processed and then written to the output file. In my case, the action-chart document is the input file and the final source code file is the output. The chart in Figure 4, page 32, shows more detail added to both the main body of the program and the subprocedure called *advance_index*. The string in the variable *work_str* is parsed from left to right in order to locate the point at which bracket characters end and the actual text begins.

In the process of passing by the action-chart characters, a nesting-level count is derived that is then used to add the proper indentation to the string before it is written to the destination file. Provisions are made to allow the number of spaces and/or tabs for the left margin to be controlled as well as the number of spaces of indentation per block level. For Pascal, I use no spaces or tabs for the left margin and two spaces per block level. For assembler, I specify one tab for the left margin and zero spaces per block level.

Some enhancements that could be made to the filter program include automatically detecting when a group of statements needs to be surrounded by *begin/end* keywords (Pascal) or curly braces (C); automatically placing semicolons on the end of statements that need them; and generating labels when working with assembler or a high-level language that uses *GOTO*s, such as Basic. With regard to processing labels, it would be good to allow for user-defined labels (e.g. *MAINLOOP*, *ERROR1*, and so on) as well as serialized ones that the filter *program would* generate when none were explicitly specified (for example, *A0001*, *A0002*, and so on). Labels can be shared by more than one action-chart construct when they coincide because of nesting or by being adjacent, and they can be placed in column 1 of the output file even when tabs or spaces are specified for the left margin.

When translating from action-chart logic to assembler, note the inverse relationship between the high-level if statement of an action-chart and the resulting assembler code, which will test and jump based on the opposite condition. Figure 5, page 32, illustrates an expression of the type if *ax* > *data1*, which is a positive logic statement, and the resulting assembler code, which is of the negative logic form. In Figure 5a, what actually needs to be stated is, "If *ax* is *not* greater than *data1* then skip the next two instructions." To keep the association between a program's action chart and its source code direct, you can use a macro like that shown in Figure 5b. The assembler code in Figure 5c would then read "Flow through if *ax* is greater than *data1*, else skip to the label *x1*." The fourth macro parameter, *else*, is added merely for readability.

Translating Flowcharts to Action Charts and Reverse Charting

When you work with code you've never seen before, or even a complex piece of code you wrote some time ago, reverse engineering it into an action chart can be a great aid to understanding it in depth. When working with a high-level language and well-structured code, going from source code to an action chart is relatively easy. When a clear structure does not exist, however, or when it is not immediately apparent (as can often be the case with assembler), an intermediate representation in flowchart format may be necessary. The fact that flowcharts do not impose any particular structure can actually be of benefit.

In working to reverse chart existing code, it is not uncommon to find logic that cannot be translated directly into an action chart. In such cases I often sketch a flowchart of the code, including only enough detail to match the chart to the source code (for example, just significant line numbers or labels and minimal comments). Next, I translate this into an action-chart skeleton, carrying over the minimal comments as a link with the original. Using this link, I can then look back at the original source code and fill in the details. In many cases you will need to transform the logic to achieve a structure that will fit properly within the rules of action diagrams. When you find sections of code being entered or exited via multiple program paths, you can often isolate the shared code and convert it into a subprocedure that can then be called from each path in a structured manner.

Conclusions

Although using macros with your present editor lets you begin utilizing this tool immediately, an editor designed especially for action-chart development could offer many powerful features and conveniences. Applying these methods in an editor with a more sophisticated macro facility should also offer many interesting possibilities. In addition, more sophisticated filters and translation programs that produce action-chart documents from existing source code will make the work of programming much easier.

I see a growing movement toward a widespread application of software engineering tools. Most of the tools now available, however, focus on the design and verification of data structures to the exclusion of procedural logic development. Action charts and the implementation of common-keystroke macros within an editor are a way of making a certain class of software engineering tools immediately available to anyone. By encouraging the use of such tools, not only will the quality of software development improve but the range of available tools will increase as well.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Ada For Pascal Programmers

Getting started in Ada is relatively easy because of its surface resemblance to Pascal. Mastering Ada is much harder, however, because of its size and complexity.

by K. N. King

Pascal was the starting point for the design of Ada, and Ada programs—at least superficially—resemble Pascal programs. Ada is not a superset of Pascal, however; as we'll see later, Ada is missing at least two of Pascal's features. (Incidentally, when I refer to Pascal, I mean ISO Standard Pascal. Some Pascal compilers incorporate a few Ada-like extensions.)

Although Ada is based on Pascal, Ada is a much larger language. Not only does Ada have many features but it also allows these features to be combined in a virtually unlimited number of ways. Getting started in Ada is relatively easy because of its surface resemblance to Pascal. Mastering Ada is much harder, however, because of its size and complexity.

Like Pascal, Ada provides a rich collection of data types, including array, record, and pointer types (pointer types are called access types in Ada), although Ada's array and record types have additional options not available in Pascal. Ada's control structures (the *if*, *case*, *while*, and *for* statements) are similar to Pascal's. Instead of Pascal's *repeat* statement, however, Ada has the more general *loop* statement. Like Pascal, Ada provides both procedures and functions, known collectively as subprograms.

Ada, like Pascal, is a strongly typed language. Every object must have a fixed, declared type, and—in general—types must match when objects are combined in expressions. Pascal has a handful of exceptions to the rule of strong typing. In particular, Pascal allows the mixing of integer and real values in expressions; an integer value is automatically converted to type *real* before being combined with a real value. Ada is stricter than Pascal; values of types *Integer* and (*Float* corresponds to Pascal's *real* type) may not be combined in an expression without explicit type conversion.

Ada combines the security of Pascal with the power of C. Ada compilers are required to perform a great deal of type checking in an effort to catch potential errors at compile time. Yet, programmers can override type checking when it hinders access to the underlying machine.

In this article, I'll discuss several of Ada's more significant features, including overloading, packages, separate compilation, private types, exceptions, and generics. Because space limitations preclude detailed discussion of some important features—including tasking and representation specifications—I'll mention them briefly at this point.

Unlike Pascal and C (and most other common languages, for that matter), Ada provides built-in support for tasking. An Ada program may consist of many tasks, each executing—at least from the programmer's point of view—asynchronously and in parallel. Ada provides a novel mechanism (the rendezvous) for tasks to synchronize with each other and exchange information; tasks can also communicate through shared variables. Ada's tasking features include support for real-time applications.

Like Pascal, Ada normally hides machine-level details from the programmer. Unlike Pascal, however, Ada allows access to these details when absolutely necessary. In particular, Ada programs may locate a variable at a particular address or specify its size, perform unchecked type conversion, and do a host of other things that depend on knowledge of the target machine (the machine on which the program will run) and/or the particular Ada system being used.

Surprisingly for such a large language, Ada lacks two features of Pascal: support for sets and the ability to pass a subprogram as a parameter to another subprogram. Built-in support for sets is not as important in Ada as in Pascal because Ada programmers can always create a package that provides a set type and a collection of set operations. The advantage of Ada's do-it-yourself approach is that you can tailor the set type to your needs—in particular, you can choose the maximum size of the set and the method in which it is implemented.

Pascal vs. Ada: An Example

When used to write simple programs, Ada is not terribly different from Pascal. There are, however, a few Ada characteristics that you should keep in mind, especially if you are an

K.N. King is an associate professor of mathematics and computer science at Georgia State University, Atlanta, GA 30303. He is the author of Modula-2: A Complete Guide.

experienced Pascal programmer. For one thing, input/output procedures are not built into Ada as they are in Pascal. To perform I/O, an Ada program can use one of the standard I/O packages that is provided with every Ada system.

Of Ada's I/O packages, the most often used is *Text_IO*, which supplies subprograms for reading and writing textual information. *Text_IO* subprograms include *Get*, *Put*, *Skip_Line*, and *New_Line*. *Get* reads one item of data, while *Put* writes one item. (As we'll see later, *Get* and *Put* are "overloaded"; *Text_IO* actually contains a number of subprograms with these names.) *Skip_Line* skips any characters that remain on the current input line. *New_Line* ends the current line of output and advances to the next line.

For a program to gain access to the subprograms in *Text_IO*, it must name the package in a *with* clause. A *with* clause by itself allows access to the subprograms in *Text_IO*, but each call of a subprogram must include the package name. For example, a call of *Put* would look like this:

```
Text_IO.Put(Ch);
```

Following a *with* clause by a *use* clause provides direct access to the names in *Text_IO*. If a *use* clause is present, calls of *Put* don't have to mention *Text_IO*:

```
Put(Ch);
```

An Ada main program can be either a procedure or a function (usually the former). Procedure names, like all other identifiers in Ada, may contain underscores. Like Pascal, Ada is not sensitive to the case of letters in an identifier.

In Ada, the semicolon is a statement terminator; it must appear at the end of every statement. In Pascal, the semicolon is a statement separator; it separates statements rather than terminating them. Ada strings are enclosed in double quotes; Pascal requires single quotes. Ada comments begin with the - - symbol and continue to the end of the line.

Example 1, this page, shows a Pascal program that counts the number of times each letter (either upper or lowercase) occurs in the input stream. Example 2, this page, is an equivalent Ada program.

Example 2 reveals one of Ada's quirks. Subprograms for reading and writing integers are located not in *Text_IO* itself, but in the generic package *Integer_IO*, which is nested inside *Text_IO*. Lines 4 and 5 instantiate this package and provide direct access to its subprograms. I'll discuss generic packages and instantiation in more detail later. Ada arrays are similar to Pascal arrays. Notice in line 6, however, that Ada requires parentheses instead of square brackets to enclose array bounds. Also, Ada allows the use of aggregates to initialize arrays. The aggregate (*others* => *0*) indicates that all components of the *Counts* array should be assigned the value 0. In line 7, notice that Ada's character type is named *Character* instead of *char*, and in line 9, that Ada's *while* statement has the form *while ... loop ... end loop*. Any number of statements may appear between the words *loop* and *end loop*. The function *End_Of_File* is from the *Text_IO* package.

We can omit the parentheses around *'a' <= Ch* and *Ch <= 'z'* in line 11 because the *and* operator has lower precedence than the <= operator. The parentheses are mandatory in Pascal, which has only four levels of operator precedence. Ada

```
program CountLetters(input, output);
  { counts occurrences of letters in the input stream }
  var Counts: array ['a'..'z'] of integer;
      Ch: char;
begin
  for Ch := 'a' to 'z' do
    Counts[Ch] := 0;
  while not eof do
    begin
      read(Ch);
      if ('a' <= Ch) and (Ch <= 'z') then
        Counts[Ch] := Counts[Ch] + 1
      else if ('A' <= Ch) and (Ch <= 'Z') then
        begin
          Ch := chr(ord(Ch) - ord('A') + ord('a'));
          Counts[Ch] := Counts[Ch] + 1
        end
    end;
  for Ch := 'a' to 'z' do
    writeln(Ch, Counts[Ch]:6)
end.
```

Example 1: *A Pascal program that counts occurrences of letters*

```
 1. with Text_IO; use Text_IO;
 2. procedure Count_Letters is
 3.    -- counts occurrences of letters in the input stream
 4.    package Int_IO is new Integer_IO(Integer);
 5.    use Int_IO;
 6.    Counts: array ('a'..'z') of Integer := (others => 0);
 7.    Ch: Character;
 8. begin
 9.    while not End_Of_File loop
10.      Get(Ch);
11.      if 'a' <= Ch and Ch <= 'z' then
12.        Counts(Ch) := Counts(Ch) + 1;
13.      elsif 'A' <= Ch and Ch <= 'Z' then
14.        Ch := Character'Val(Character'Pos(Ch) -
15.                            Character'Pos('A') +
16.                            Character'Pos('a'));
17.        Counts(Ch) := Counts(Ch) + 1;
18.      end if;
19.    end loop;
20.    for Ch in 'a'..'z' loop
21.      Put(Ch);
22.      Put(Counts(Ch), 6);
23.      New_Line;
24.    end loop;
25. end Count_Letters;
```

Example 2: *An Ada program that counts occurrences of letters in the input stream*

PASCAL COMPARISON
(continued from page 35)

has six levels of precedence, with the logical operators *and*, *or*, and *xor* at a lower level than the relational operators <, <=, >, >=, =, and /= (not equal). In lines 11–18, notice that Ada's *if* statement has the form *if* ... *then* ... {*elsif* ... *then* ...} [*else* ...] *end if*, where the braces mean that there may be any number of *elsif* clauses (including none) and the square brackets mean that the *else* clause may be omitted.

In Ada, entities (including objects and types) may have attributes. One of the attributes of the *Character* type is the function *Character'Pos*, used on lines 14–16, which is similar to Pascal's *ord* function. Another attribute of *Character* is *Character'Val*, which is equivalent to Pascal's *chr* function. The *for* statement in line 20, like the *while* statement, is a special case of the *loop* statement. The *for* statement specifies a loop parameter (*Ch*, in this case) and a range of values for the loop parameter to assume ('a'..'z'). A loop parameter is not a normal variable (although it may have the same name as an ordinary variable): It need not be declared, and it is not visible outside the loop. The second parameter to the call of *Put* (line 22) indicates that *Counts(Ch)* should be written using at least six characters; if *Counts(Ch)* does not require six characters, extra space will precede it. This parameter is a default parameter; it may be omitted, in which case it assumes a default value. Ada subprograms may have any number of default parameters.

Overloading

One of Ada's more unusual features is its tolerance for overloading (assigning different meanings to the same symbol). Pascal has a small amount of built-in overloading (for example, the symbol + represents integer addition, real addition, and set union) but provides no support for programmer-defined overloading. Ada allows overloading of several kinds of entities, including subprogram names. When it encounters a call of an overloaded subprogram, an Ada compiler must decide which subprogram is actually being called. It makes this decision by examining the number of actual parameters and their types.

Text_IO's *Get* and *Put* procedures are heavily overloaded. For example, consider the *Put* procedures shown in Example 3, below. If a call of *Put* has two parameters, the first of type *File_Type* and second of type *Character*, then the compiler deduces that this is a call of the first version of *Put*. If there is only one parameter and its type is *Character*, then this is a call of the second version of *Put*.

One of the advantages of overloading is that similar subprograms can be given the same name so that programmers have fewer names to remember. If abused, however, overloading can make programs hard to read.

Packages

An Ada program consists of one or more compilation units. One of these must be a subprogram that serves as the main program; the other units may be subprograms or packages. A package is a collection of other Ada entities, including constants, types, variables, and subprograms. Each package has two parts: a specification and a body. The specification lists information to be made available to the rest of the program, and the body contains information to be hidden from the rest of the program.

A package may be nothing more than a collection of object and/or type declarations. Example 4, on page 37, shows a package called *Length_Conversions* that contains declarations of four useful constants. This package needs no body; there is no information to be hidden.

The entities declared in a package specification are available for use by other compilation units (its clients).

```
procedure Put(File: File_Type; Item: Character);
procedure Put(Item: Character);
procedure Put(File: File_Type; Item: String);
procedure Put(Item: String);
```

Example 3: *Overloading the Put procedure*

First, however, each client must mention the package in a *with* clause. Example 5, page 38, shows a program named *Convert_To_Meters* that uses some of the constants in *Length_Conversions*. Notice that *Convert_To_Meters* gains access to the constants in *Length_Conversions* in the same way as it gains access to the subprograms in *Text_IO*: through *with* and *use* clauses.

A collection of related subprograms also makes a natural package. Example 6, page 38, shows the specification of a package (*Angle_Conversions*) that provides functions for converting from degrees to radians and vice versa. When a package supplies subprograms, only the headings of the subprograms appear in the specification of the package. Clients of the package need to know how to call the subprograms but don't need to know how the subprograms work.

The *Angle_Conversions* package, unlike *Length_Conversions*, must have a body. The body contains full declarations of the subprograms in the package. Example 7, page 39, shows the body of *Angle_Conversions*. Notice that an Ada function terminates by executing a *return* statement; the expression that follows the word *return* is the value that the function returns.

A package may contain hidden data structures. To show how such a data structure might arise, we'll write a small program that reverses a string entered by the user. Reversing a string is easy to do using a stack (a last-in, first-out data structure): As the program reads the characters, it pushes them onto the stack; when all characters have been read, the program pops characters from the stack, writing each character as it is popped. Let's write a package that provides subprograms (representing operations on the stack) while hiding the stack itself inside the body of the package.

We will need three stack operations: *Push* (push a character onto the stack), *Pop* (retrieve the top stack element, then pop the stack), and *Is_Empty* (determine whether the stack is empty). Example 8, page 39, shows a package named *Char_Stack* that provides these operations. A more useful stack package might include other operations (finding the top stack element without popping, testing whether the stack is full, and so forth); for simplicity, we'll limit ourselves to three operations.

Example 8 illustrates an aspect of Ada that you have not seen in previous examples. Instead of Pascal's value and variable (*var*) parameters, Ada provides three parameter modes: *in*, *out*, and *in out*. A formal parameter is declared to have mode *in* if it represents a value to be supplied to the subprogram; an *in* parameter cannot be changed by the subprogram. An *out* parameter represents a

```
package Length_Conversions is
  Feet_To_Meters: constant := 0.3048;
  Inches_To_Centimeters: constant := 2.54;
  Miles_To_Kilometers: constant := 1.6093;
  Yards_To_Meters: constant := 0.9144;
end Length_Conversions;
```

Example 4: *The specification of the Length_Conversions package*

PASCAL COMPARISON
(continued from page 37)

variable that will be assigned a value by the subprogram. An *in out* parameter represents a variable that can be both read and modified by the subprogram. The default mode is *in*. The parameter *X* to *Push* has mode *in* because *Push* needs only to read *X*, not modify it. The parameter *X* in *Pop* has mode *out* because *Pop* will store into *X* the value popped from the stack.

The body of *Char_Stack* (see Example 9, page 40) contains declarations of the objects that make up the hidden stack as well as declarations of the subprograms that operate on the stack. Notice that the value of *Top_Of_Stack* is constrained to lie between 0 and *Stack_Size*; this is similar to a Pascal subrange. For now, ignore the possibility that *Top_Of_Stack* might exceed *Stack_Size* or fall below 0; I'll address the issue of error handling later. Example 10, page 40, shows a main program that uses the *Char_Stack* package to reverse a string.

```
with Text_IO, Length_Conversions;
use Text_IO, Length_Conversions;
procedure Convert_To_Meters is
  package Int_IO is new Integer_IO(Integer);
  use Int_IO;
  Feet: Integer;
begin
  Put("Enter a measurement in feet: ");
  Get(Feet);
  Skip_Line;
  Put("The equivalent measurement in meters is: ");
  Put(Integer(Float(Feet)*Feet_To_Meters), 1);
  New_Line;
end Convert_To_Meters;
```

Example 5: *A program that uses the Length_Conversions package*

```
package Angle_Conversions is
  function Degrees_To_Radians(Degrees: Float) return Float;
  function Radians_To_Degrees(Radians: Float) return Float;
end Angle_Conversions;
```

Example 6: *The specification of the Angle_Conversions package*

Separate Compilation

The various subprograms and packages that make up a program may be kept in separate files and compiled separately, and the specification of a package may be compiled separately from the body of the package. The order of compilation is important, however. The specification of a package must be compiled before the body of the package and before all clients of the package. (For example, we must compile the specification of *Char_Stack* before we compile the body of *Char_Stack* and before we compile *Reverse_String*.) This rule ensures that the compiler will have access to the information in the package specification when the body (and the clients) are compiled. Restricting the order of compilation allows an Ada compiler to perform full type checking across the boundaries of compilation units.

When the body of a package is changed and recompiled, the rest of the program need not be recompiled, provided that the specification of the package did not change. Turbo Pascal 4.0 provides a package like feature known as a unit, but units lack some of the benefits of Ada packages. A Turbo Pascal unit has an "interface" section and an "implementation" section, but both sections must be kept in the same file. As a result, when the implementation of a unit changes, all clients of the unit must be recompiled, even if the interface has not changed.

Private Types

The *Char_Stack* package is an example of a reusable software component. Although it was written for use in the *Reverse_String* program, *Char*

```
package body Angle_Conversions is

  Two_Pi: constant := 2.0 * 3.14159;

  function Degrees_To_Radians(Degrees: Float) return Float is
  begin
    return Two_Pi * Degrees / 360.0;
  end Degrees_To_Radians;

  function Radians_To_Degrees(Radians: Float) return Float is
  begin
    return 360.0 * Radians / Two_Pi;
  end Radians_To_Degrees;

end Angle_Conversions;
```

Example 7: *The body of the Angle_Conversions package*

```
package Char_Stack is

  procedure Push(X: Character);
    -- pushes X onto the stack

  procedure Pop(X: out Character);
    -- stores the top stack element into X, then pops the stack

  function Is_Empty return Boolean;
    -- returns True if the stack is empty, False otherwise

end Char_Stack;
```

Example 8: *The specification of the Char_Stack package*

PASCAL COMPARISON
(continued from page 39)

Char_Stack is general enough to be used in other programs as well. The usefulness of *Char_Stack* is limited, however, because it hides a single stack of characters. If we're writing a program that uses two or more stacks, we want a stack type that we can use to declare as many stack variables as needed.

We can easily modify the package to provide a *Char_Stack* type along with the *Push*, *Pop*, and *Is_Empty* subprograms. Example 11, page 43, shows the specification of the new package, which we'll call *Char_Stacks*. A client of *Char_Stacks* can declare one or more variables of type *Char_Stack* and use the *Push*, *Pop*, and *Is_Empty* subprograms to operate on these variables:

```
S: Char_Stack;
...
Push(S, Ch);
```

There's a problem with this definition of the *Char_Stack* type, however: a *Char_Stack* variable is actually a record, and the client has full access to the components of this record. There is nothing to prevent the client from accessing—or even modifying—the components of a *Char_Stack* variable. For example, the client might bypass the *Push* procedure as follows:

```
S: Char_Stack;
...
S.Top_Of_Stack :=
                    S.Top_Of_Stack + 1;
S.Stack_Array(S.Top_Of_Stack) :=
                    Ch;
```

This practice is dangerous because there is no guarantee that the stack will remain consistent. For example, the client might increment *Top_Of_Stack* without storing into *Stack_Array* or modify a component of *Stack_Array* that is not at the top of the stack.

Another problem caused by direct manipulation of the stack is that the client becomes dependent on a single representation of the stack. We would like to retain the option of representing the stack in some other form. For example, if stack overflow is a problem, we might rewrite the *Char_Stacks* package so that a *Char_Stack* variable is a pointer to a linked list of records. If clients have used only the *Push*, *Pop*, and *Is_Empty* subprograms to access *Char_Stack* variables, then the clients will not need to be changed.

The designers of Ada recognized the problems that can occur when clients have access to the representation of a type such as *Char_Stack*. To prevent such access, we can declare *Char_Stack* to be a private type. If *Char_Stack* is a private type, clients cannot take advantage of the fact that a *Char_Stack* variable is really a record; clients can supply a *Char_Stack* variable to a call of *Push*, *Pop*, or *Is_Empty* but may not directly examine or change its components.

Example 12, page 43, shows the appearance of *Char_Stacks* when we make *Char_Stack* a private type. Notice that the full declaration of Char_Stack still appears in the package specification but is "hidden" at the end in a special section known as the private part. Information that appears in the private part of a package specification is visible to the compiler but not to clients of the package. Clients know only that *Char_Stack* is a type, not that it is a record type.

Example 13, page 44, shows the body of the *Char_Stacks* package. The body does not depend on whether or not *Char_Stack* is a private type. Example 14, page 44, shows the *Reverse_String* program after it has been modified to use the *Char_Stacks* package.

```
package body Char_Stack is

   Stack_Size: constant := 100;  --maximum size of stack
   Stack_Array: array (1..Stack_Size) of Character;
   Top_Of_Stack: Integer range 0..Stack_Size := 0;

   procedure Push(X: Character) is
   begin
     Top_Of_Stack := Top_Of_Stack + 1;
     Stack_Array(Top_Of_Stack) := X;
   end Push;

   procedure Pop(X: out Character) is
   begin
     X := Stack_Array(Top_Of_Stack);
     Top_Of_Stack := Top_Of_Stack - 1;
   end Pop;

   function Is_Empty return Boolean is
   begin
     return Top_Of_Stack = 0;
   end Is_Empty;

end Char_Stack;
```

Example 9: *The body of the Char_Stack package*

```
with Text_IO, Char_Stack;
use Text_IO, Char_Stack;
procedure Reverse_String is
  Ch: Character;
begin
  Put("Enter string to be reversed: ");
  while not End_Of_Line loop
    Get(Ch);
    Push(Ch);
  end loop;
  Skip_Line;

  Put("The reversal is: ");
  while not Is_Empty loop
    Pop(Ch);
    Put(Ch);
  end loop;
  New_Line;
end Reverse_String;
```

Example 10: *A program that uses the Char_Stack package to reverse a string*

PASCAL COMPARISON
(continued from page 40)

Exceptions

An exception is an error (division by 0, for example) or other unusual condition that occurs during program execution. Ada provides support for naming an error condition (declaring an exception), for responding to an exception (handling the exception), and for causing an exception to occur (raising the exception).

The following exceptions are predefined in Ada: *Constraint_Error*, *Numeric_Error*, *Program_Error*, *Storage_Error*, and *Tasking_Error*. Of these, the most common are *Constraint_Error*, which is raised when any kind of constraint is violated (often a subscript out of range) and *Numeric_Error*, which is raised by conditions such as overflow and division by 0. Although there are only five built-in exceptions, programmers may declare others as needed.

What happens when an exception is raised during the execution of a program? If it fails to handle the exception, the program terminates. (When an exception is not handled, an Ada system normally displays the name of the exception and line number on which it was raised.) To be able to handle an exception, we must include extra code indicating what to do if a certain exception is raised at a particular point in the program. Here's one way to define an exception handler:

```
begin
statements in which exception might
be raised
exception
   when
exception_name => code for exception handler
end;
```

If the named exception is raised in one of the statements between *begin* and *exception*, control is transferred to the handler; after it has executed, control is transferred to the statement just after the word *end*. If no exception is raised during execution of the statements between *begin* and *exception*, then everything between *exception* and *end* is skipped.

Consider the *Push* procedure in the *Char_Stacks* package. When *S.Top_Of_Stack* is equal to *Stack_Size*, incrementing *S.Top_Of_Stack* will raise the *Constraint_Error* exception, because *S.Top_Of_Stack* is constrained to be in the range 0..*Stack_Size*. When an exception is raised and the current subprogram doesn't handle it, the subprogram terminates and the exception is raised in the calling subprogram. Since *Push* doesn't have a handler for the *Constraint_Error* exception, the caller of *Push* will get an opportunity to handle the exception. Unfortunately, the client must have some information about how the stack is implemented in order to know what exception will be raised. If the stack were implemented as a linked list, stack overflow might be indicated by the *Storage_Error* exception instead of *Constraint_Error*.

Because of problems such as this, Ada allows the declaration of new exceptions. If an exception is declared in the specification of a package, it becomes available to clients of the package. In the case of the *Char_Stacks* package, there are two errors that could occur during the use of the package: overflow (attempting to push when the stack is full) and underflow (attempting to pop when the stack is empty). Therefore, we declare exceptions named *Overflow* and *Underflow* in the specification of *Char_Stacks* (see Example 15, page 46).

When *Push* detects that the stack is full, it raises the *Overflow* exception, using the Ada *raise* statement. Similarly, when *Pop* detects that the stack is empty, it raises the *Underflow* exception (see Example 16, page 48).

Clients of *Char_Stacks* can now

```
package Char_Stacks is

   Stack_Size: constant := 100;
   type Array_Type is array (1..Stack_Size) of Character;
   type Char_Stack is
      record
         Stack_Array: Array_Type;
         Top_Of_Stack: Integer range 0..Stack_Size := 0;
      end record;

   procedure Push(S: in out Char_Stack; X: Character);
      -- pushes X onto stack S

   procedure Pop(S: in out Char_Stack; X: out Character);
      -- stores the top element of S into X, then pops S

   function Is_Empty(S: Char_Stack) return Boolean;
      -- returns True if S is empty, False otherwise

end Char_Stacks;
```

Example 11: *The specification of the Char_Stacks package; Char_Stack is an ordinary type*

```
package Char_Stacks is

   type Char_Stack is private;

   procedure Push(S: in out Char_Stack; X: Character);
      -- pushes X onto stack S

   procedure Pop(S: in out Char_Stack; X: out Character);
      -- stores the top element of S into X, then pops S

   function Is_Empty(S: Char_Stack) return Boolean;
      -- returns True if S is empty, False otherwise

private
   Stack_Size: constant := 100;
   type Array_Type is array (1..Stack_Size) of Character;
   type Char_Stack is
      record
         Stack_Array: Array_Type;
         Top_Of_Stack: Integer range 0..Stack_Size := 0;
      end record;
end Char_Stacks;
```

Example 12: *The specification of the Char_Stacks package; Char_Stack is a private type*

PASCAL COMPARISON
(continued from page 43)

provide handlers for the *Overflow* and *Underflow* exceptions. Example 17, page 48, shows *Reverse_String* modified to handle the *Overflow* exception. (If the stack overflows, *Reverse_String* simply ignores the remaining input. Notice the use of the Ada *null* statement—a "dummy" statement that has no effect.)

Generics

The *Char_Stacks* package is fine if we need only stacks whose elements are characters. What if we need stacks whose elements are integers, or real numbers, or records? One possibility would be to make a copy of the *Char_Stacks* package, replacing each occurrence of *Character* by some other type. This solution is not completely satisfactory. Not only is it a lot of work but it would also cause problems if we ever needed to change the package (to add additional subprograms, say, or to fix a bug, or to change the way the stack is represented): we would have to track down every copy of the package and change each one individually.

The designers of Ada anticipated this problem and provided a special feature to solve it. When we write a package, Ada allows us to omit certain information; we supply the missing information later, when the package is used. In the case of the *Char_Stacks* package, we would like to omit the type of the stack elements; this would allow us to write a completely general stack package that could be used with elements of any type. Such a package is said to be generic.

```
package body Char_Stacks is

  procedure Push(S: in out Char_Stack; X: Character) is
  begin
    S.Top_Of_Stack := S.Top_Of_Stack + 1;
    S.Stack_Array(S.Top_Of_Stack) := X;
  end Push;

  procedure Pop(S: in out Char_Stack; X: out Character) is
  begin
    X := S.Stack_Array(S.Top_Of_Stack);
    S.Top_Of_Stack := S.Top_Of_Stack - 1;
  end Pop;

  function Is_Empty(S: Char_Stack) return Boolean is
  begin
    return S.Top_Of_Stack = 0;
  end Is_Empty;

end Char_Stacks;
```

Example 13: *The body of the Char_Stacks package*

```
with Text_IO, Char_Stacks;
use Text_IO, Char_Stacks;
procedure Reverse_String is
  S: Char_Stack;
  Ch: Character;
begin
  Put("Enter string to be reversed: ");
  while not End_Of_Line loop
    Get(Ch);
    Push(S, Ch);
  end loop;
  Skip_Line;

  Put("The reversal is: ");
  while not Is_Empty(S) loop
    Pop(S, Ch);
    Put(Ch);
  end loop;
  New_Line;
end Reverse_String;
```

Example 14: *A program that uses the Char_Stacks package to reverse a string*

Validation: Ada's Greatest Strength Or Weakest Link?

Among the stated design goals for the Ada language were a desire: to reduce software development and maintenance costs, to provide portability of both software and programmers, and to encourage sound software engineering principles. To achieve these goals, Ada proponents established a set of validation tests that are supposed to ensure that compilers comply with a standard as well as to prevent the introduction of Ada subsets, supersets, or dialects.

In theory, the validation process is a good way to achieve these goals and there's no reason why, in practice, validation shouldn't be straightforward to implement. But there's often a big difference between theory and practice, and it's been the validation process itself that has scared off some compiler developers, thereby stifling acceptance of the language. To illustrate this point, what follows is a description of the experiences of one developer who wasn't scared by validation. — eds.

Ada Validation: The Diary of a Vendor

by James Stewart

At R.R. Software, we have conducted almost a dozen validations, both for our own compilers and for those we've made for other companies. This description of the validation process is tendered with admiration and respect for the participants, whom we trust can laugh as well as they work.

Getting Started

First, get an Ada compiler. Although that may sound easy, it takes the average developer more than 16 person years and two million dollars to accomplish.

Next, get the current validation suite from the government. Unless you know where to go, this could be almost as difficult as getting a compiler. The place to start is the Ada Joint Program Office (AJPO) in Washington D.C. which will direct you to the Ada Validation Facilities (AVF). The AVF will send you the current Ada Compiler Validation Capability (ACVC) if you send the ACVC the appropriate media for your computer and indicate that you are planning to validate.

Now you're ready for the real fun . . .

Scheduling a Validation

To validate an Ada compiler, you must contract with your AVF. As it is working for the government, you should be prepared to wait . . . and wait . . and wait. Prior to scheduling your validation, you should do two things: test your compiler against the ACVC and make sure the ACVC you have will still be in effect when you want to validate. We will assume that your compiler passes all the applicable tests (this is an all-or-nothing test). As the ACVC gets revised each year, you have to be careful to ensure you're testing with the current model (ACVC 1.10, currently).

If you have gotten this far, prepare your letter to the AVF for a contract to validate; it should indicate what computer environments you are using, what compiler version you are testing, and what validation suite you expect to validate under. You should also indicate when you will send the prevalidation results and what date you want for the on-site validation. Last, you should give the AVF an estimate of how long you expect the actual validation to take (you should have a rough idea from running the tests for prevalidation).

Contracting for Validation

Having sent the AVF your letter, you will receive a letter confirming the subject matter of your letter. You will also receive a fairly short and simple government contract that details the actions of both parties, when the prevalidation and validation will occur, and what you must pay for this process. The cost of validation depends on several factors: the speed of your environment, the number of validations for which you contract, the number of prevalidations you desire (this lowers your risk at validation but is costly) as well as a plethora of more subtle factors. The important point to remember is that you will pay the indicated cost prior to commencing the validation process; be prepared to write the check if you're in a hurry!

The Prevalidation Process

You should already have tested your compiler against the ACVC in-house; now you need to submit the results, in printed listings and media form, to the AVF. It will check the results and inform you when it thinks there are any problems. If you disagree, the question will be submitted to the Ada Validation Office (AVO), which decides the issue. This process can run anywhere from 45 days to three months, so grab a good book to read during this period. If you have done the job correctly, this will be a relatively quick and easy process.

The Actual Validation

The appointed day arrives and your validation team shows up, ready to test the compiler. The people on the team will need a room to grade the tests after completion. They'll load the ACVC and start it running; from that point on, you will follow their directions and interact with them at least twice each day. Our own experiences with the Wright-Pattterson AVF have always been very pleasant; the people on the team have always been very helpful and professional, which is quite a relief during a tense period such as validation. When the process is completed, they go over the final Validation Summary Report (VSR) with you, telling you that the results are subject to review and sign-off by the AVO and AJPO. Therefore you can look forward to another wait for the actual certificates, usually six to eight weeks.

Post-Validation

You now have the final VSR and your certificates, so you think it's all over until you look at the expiration date on the certificates to see that this will only last a year. This three to six month process is a yearly event with a tougher test *each time*.

James Stewart is general manager of R.R. Software Inc. and has participated in the Ada field for more than six years. He holds degrees from the University of Texas and the University of Wisconsin.

PASCAL COMPARISON
(continued from page 45)

Example 18, on page 51, shows the specification of a generic *Stacks* package. The word *generic* at the beginning identifies *Stacks* as a generic package. The next line specifies that the type *Element* will be supplied later, when the package is used. (*Element* is not an ordinary private type; it is a *generic type* parameter. The words *is private* simply mean that *Stacks* must treat *Element* as a private type; it cannot assume that *Element* is a record type, for example.)

```
package Char_Stacks is

  type Char_Stack is private;

  procedure Push(S: in out Char_Stack; X: Character);
    -- pushes X onto stack S; raises Overflow if S is full

  procedure Pop(S: in out Char_Stack; X: out Character);
    -- stores the top element of S into X, then pops S
    -- raises Underflow if S is empty

  function Is_Empty(S: Char_Stack) return Boolean;
    -- returns True if S is empty, False otherwise

  Overflow, Underflow: exception;

private
  Stack_Size: constant := 100;
  type Array_Type is array (1..Stack_Size) of Character;
  type Char_Stack is
    record
      Stack_Array: Array_Type;
      Top_Of_Stack: Integer range 0..Stack_Size := 0;
    end record;
end Char_Stacks;
```

Example 15: *The specification of the Char_Stacks package with exceptions added*

The body of *Stacks* is identical to the body of *Char_Stacks* (Example 16), except that *Stack* replaces *Char_Stack* and *Element* replaces *Character*.

A generic package is not an ordinary package; it is a template from which packages can be created by a process called instantiation. Instantiating a generic package requires filling in the blanks that were left when the package was written. Before it can use the entities in the *Stacks* package, a client must first instantiate the package by specifying what *Element* is.

Example 19, page 51, shows how this would be done in the *Reverse_String* program. *Reverse_String* first instantiates the generic *Stacks* package by specifying that *Character* be substituted for *Element*; the resulting package is given the name *Char_Stacks* (any name will do). The *use* clause on the following line allows *Reverse_String* to access *Push*, *Pop*, and *Is_Empty* directly, without having to mention *Char_Stacks* each time.

You have seen instantiation of a generic package before. Examples 2 and 5 contain the following lines:

```
package Int_IO is
            new Integer_IO(Integer);
use Int_IO;
```

The purpose of these lines is to instantiate the generic package *Integer_IO*, specifying that the procedures in *Integer_IO* will be used to read and write *Integer* values (as opposed to *Short_Integer* or *Long_Integer* values). The name of the instantiated package is chosen by the programmer; I use the name *Int_IO*. The *use* clause that follows makes the names of the procedures in *Int_IO* directly visible.

Ada's Problems

Ada is not the utlimate programming language. Like every other programming language, it has both advantages and disadvantages. I've discussed some of Ada's attractive features; now let's take a look at its problems. Some of Ada's powerful features have hidden disadvantages. Exception handling, for example, may

mask errors; unforeseen errors may be handled without the programmer's knowledge. Overloading and use of default parameters can make programs hard to read.

Problems of compile- and run-time efficiency have plagued Ada since the beginning. (A couple of years ago I heard that a well-known Ada compiler took 90 minutes to compile a 36-line program, then took another 45 minutes to link. And this was on a VAX 11/780 with no other jobs running!) Current compilers exhibit decent compilation speed, although few are likely to challenge Turbo Pascal. Slow compilation can largely be attributed to several of Ada's more complex features, including overloading and generics. Because of these features, Ada compilers will probably always be slower than compilers for Pascal, C, and other popular languages.

The below-par run-time performance of Ada programs is due to two factors. One is the extensive checking that Ada requires; although much of this checking can be done during compilation, a substantial part must be postponed until the program is run. Turning off these checks improves execution speed but may hide bugs and cause timing problems. Another factor is the stringent requirements of Ada validation (see the sidebar on page 45), which encourage compiler developers to devote substantial effort to validation at the expense of code generation and optimization.

Ada can be a deceptive language—its true complexity is not always apparent to beginners. Because of Ada's surface similarity to Pascal, programmers quickly progress to the stage of being able to read Ada programs. Writing Ada programs of small-to-moderate size is harder, but should not present too great an obstacle to most programmers. Getting to this stage may give the programomer a false sense of security, however. True mastery of Ada is extremely difficult because of the size of the language, the number of language features (some only rarely used), and the complex ways in which they interact. Few programmers will ever understand Ada at the level they understand smaller languages such as Pascal or C. One of the dangers of Ada is that project managers, misled by Ada's similarity to Pascal, will assume that Ada is not much harder to master than Pascal. Once committed to Ada, however, they discover that no one on the staff really understands it well enough to answer hard questions.

The complexity of Ada runs counter to the thinking of language designers who believe that programming languages should be kept small. Niklaus Wirth, the creator of Pascal and Modula-2, is in this group. Modula-2 appeared at about the same time as Ada. Although it is suitable for the same range of applications as Ada, Modula-2 is a simpler language. It contains several Ada-like features, including packages (modules in Modula-2), private types (opaque types in Modula-2), tasking (implemented using coroutines in Modula-2), and low-level access to the underlying machine. Modula-2, however, lacks some of Ada's more sophisticated (but expensive to implement) features, such as overloading, exceptions, and generics. In

```
package body Char_Stacks is

  procedure Push(S: in out Char_Stack; X: Character) is
  begin
    if S.Top_Of_Stack = Stack_Size then
      raise Overflow;
    end if;
    S.Top_Of_Stack := S.Top_Of_Stack + 1;
    S.Stack_Array(S.Top_Of_Stack) := X;
  end Push;

  procedure Pop(S: in out Char_Stack; X: out Character) is
  begin
    if S.Top_Of_Stack = 0 then
      raise Underflow;
    end if;
    X := S.Stack_Array(S.Top_Of_Stack);
    S.Top_Of_Stack := S.Top_Of_Stack - 1;
  end Pop;

  function Is_Empty(S: Char_Stack) return Boolean is
  begin
    return S.Top_Of_Stack = 0;
  end Is_Empty;

end Char_Stacks;
```

Example 16: *The body of the Char_Stacks package with exceptions added*

```
with Text_IO, Char_Stacks;
use Text_IO, Char_Stacks;
procedure Reverse_String is
  S: Char_Stack;
  Ch: Character;
begin
  Put("Enter string to be reversed: ");
  begin
    while not End_Of_Line loop
      Get(Ch);
      Push(S, Ch);
    end loop;
  exception
    when Overflow => null;    -- ignore overflow
  end;
  Skip_Line;

  Put("The reversal is: ");
  while not Is_Empty(S) loop
    Pop(S, Ch);
    Put(Ch);
  end loop;
  New_Line;
end Reverse_String;
```

Example 17: *A program that uses the Char_Stacks package to reverse a string (with exception handling added)*

PASCAL COMPARISON
(continued from page 48)

short, Modula-2 captures the essence of Ada while avoiding Ada's overwhelming complexity.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

```
generic
  type Element is private;
package Stacks is

  type Stack is private;

  procedure Push(S: in out Stack; X: Element);
    -- pushes X onto stack S; raises Overflow if S is full

  procedure Pop(S: in out Stack; X: out Element);
    -- stores the top element of S into X, then pops S
    -- raises Underflow if S is empty

  function Is_Empty(S: Stack) return Boolean;
    -- returns True if S is empty, False otherwise

  Overflow, Underflow: exception;

private
  Stack_Size: constant := 100;
  type Array_Type is array (1..Stack_Size) of Element;
  type Stack is
    record
      Stack_Array: Array_Type;
      Top_Of_Stack: Integer range 0..Stack_Size := 0;
    end record;
end Stacks;
```

Example 18: *The specification of the generic Stacks package*

```
with Text_IO, Stacks;
use Text_IO;
procedure Reverse_String is

  package Char_Stacks is new Stacks(Character);
  use Char_Stacks;

  S: Stack;
  Ch: Character;

begin
  Put("Enter string to be reversed: ");
  begin
    while not End_Of_Line loop
      Get(Ch);
      Push(S, Ch);
    end loop;
  exception
    when Overflow => null;
  end;
  Skip_Line;

  Put("The reversal is: ");
  while not Is_Empty(S) loop
    Pop(S, Ch);
    Put(Ch);
  end loop;
  New_Line;
end Reverse_String;
```

Example 19: *A program that uses the generic Stacks package to reverse a string*

Object-Oriented Dimensional Units

OOD means thinking of dimensional units in a different way

by John A. Grosberg

Object-oriented design (OOD) differs from traditional software design approaches in that OOD assumes that programs are made up of "objects" and "classes" that send "messages" to each other, instead of being made up of procedures and data. OOD emphasizes the structure of a program over its function. When taking the OOD approach, you have to develop the knack of selecting the correct objects and their relevant messages. Selecting objects is not always straightforward since an "object" may not be a physical object — often it is a mental object.

In OOD, an *object* is a programming entity that has the characteristics of both data and procedures, while a *class* is a programming construct that defines the common features of a set of objects. These features include both the set of values that a class's objects may have and the set of permissible operations on them. The operations that a class provides do the following: create an object of the class, change the value of an object, read the value of an object, and destroy an object.

Programmers who use object-oriented languages usually refer to the operations as *messages*. You can relate one class to its subclass as parent to child. The subclass *inherits* some or all of the capabilities of the parent class and may add new capabilities of its own. If the subclass substitutes an operation for one that it inherited from its parent, making the inherited operation inaccessible, then the new operation is said to *override* the inherited one. These terms are defined without reference to any programming language because you can apply the principles of OOD to any language. Of course, doing this is easier in some languages than in others. In this article, I will describe how to apply the object-oriented approach to dimensional units in Ada. Dimensional units are used in any application that has to deal with physical quantities. The article also describes how to apply OOD to abstractions in general. Abstractions typically give novice object-oriented designers the most trouble. If you want to review a more conventional approach to implementing dimensional units in Ada, see "Dimensional Data Types," in *DDJ* May 1987, by Do-While Jones.

John Grosberg is a senior staff engineer at Motorola's Government Electronic Group and can be reached at 5842 N 86th St., Scottsdale, AZ 85253.

Dimensional Units

For most applications, you must make computations based on physical formulas, such as computing circuit quantities based on Ohm's law. Any physical formula involves terms that have units, such as volts, ohms, and amps. There are only a certain number of valid ways to combine dimensioned quantities, it makes sense to multiply amps by ohms to get volts. However, it usually doesn't make sense to multiply volts by ohms.

If you make careful use of Ada language features, the compiler can help you by detecting and rejecting incorrect combinations of units. If you are just writing a small program, this is probably not important — you can avoid them or easily find them on your own. But Ada was designed for writing large programs and, when many programmers are writing code that must mesh, they find the compiler is essential in detecting as many kinds of errors as possible.

Obviously, dimensional units are not what we normally think of as physical objects: they are characteristics of physical objects. A characteristic is an abstraction. But you can think of abstractions as existing as if they were themselves objects, having

their own properties. I call abstractions like this "mental objects" because they become objects as you think of them. Thus, dimensional units are examples of mental objects.

The way to apply OOD to dimensional units is to think of each dimensional unit that your application area requires as being a class of objects. Thus, you might have a volt class or an amp class. Or, you might have a mile class, an hour class, and a miles-per-hour class. You would implement each such class as an Ada package that exports a type and related operations.

An Ada package is a means of grouping related programming entities. A package may contain, among other things, data types, variables, constants, procedures, and functions. A package may contain any number of procedures or functions that use the same name, as long as they have different argument lists. You can make these resources available to any other Ada program unit by simply naming the package in a context, or "with" clause as shown here:

with my_package;

A package consists of two parts — a specification and a body. The specification describes the interface to the package's resources and tells what the package does. The body contains the actual Ada code and shows how the package resources do what they are required to do.

The form of the package construct is shown in Example 1, this page. This construct is part of a package specification for our prototype dimension class. The package's name is *float_unit*. It represents a class called *float unit*. The collection of resources in the package includes one type, called *class*, and a limited number of functions that can operate on objects of that type. Listing One on page 94, provides the entire package, including its specification and body. Ada uses extended or "dot" notation to refer to the resources in a package. Therefore, *float_unit.class* refers to the data type called *class* defined in the *float_unit* package. Also, in the Listing two hyphens (- -) introduce a comment, which the compiler ignores.

In the package specification of Listing One, notice that the multiplication function * (asterisk), takes a dimensionless object (left) of floating-point type, multiplies it by an object of our class (right), and produces a result that is of our class. This process should happen when, for example, you multiply the dimensionless number two times the distance to a neighboring town — say, 30 miles — to obtain the round trip distance of 60 miles.

Listing One demonstrates that *float_unit* inherits some valid operators from its parent type, *float*. These operators include addition, subtraction, and assignment. In addition *float_unit* defines some new multiplication and division operators, such as the multiplication function just described. It also overrides some inherited operators because they are invalid in the context of dimensional units. An example of an invalid inherited operator is the multiplication of two objects of the same class together to obtain a result of the same class:

function "*" (left, right:class) return class;

This operator is inherited from *float* where it is natural to multiply two floating-point (dimensionless) numbers to obtain another floating-point number. But for the dimentioned numbers the operator is invalid because if *class* is, say, foot and you multiply 5 feet by 4 feet, you do not get 20 feet — you get 20 square feet. My method of overriding the invalid operator is to define the function in the specification of the package so that it overloads the inherited operator. Then in the body, I simply raise an exception if that particular function is called. This means that the error will not be detected until runtime, but at least it will be detected. Ada provides no way to detect it at compile time.

Listing One also contains two additional functions, *image* and *value*. The *image* function converts an object that is of *float_unit.class* or that is derived from *float_unit.class* into a string. The *value* function converts

```
package float_unit  is
     type class is new float;

     function"*"(left : float;
                 right: class
                ) return  class;

     function "/" (left : class;
                   right: float
                  ) return class;

     -- etc...
end float_unit;
```

Example 1: *Using the Ada package construct to represent a class*

```
with hour;  use hour;
procedure time_card is
     -- Create the objects:
     hours_worked : hour.class;
     job_1 : hour.class;
     job_2 : hour.class;

begin
     -- Give them each a value:
     job_1 := 8.0;
     job_2 := 5.5;
     hours_worked := job_1 + job_2;

end time_card;
```

Example 2: *Creating and using objects of the hour class*

(continued from page 53)

a string that has the proper syntax into an object of the class. The *image* and *value* functions are the inverse of each other. They provide a way to go between text and *float_unit.class*. They also serve to decouple the class packages from input-output classes. (We will discuss object coupling later in this article.) All classes derived from *float_unit* inherit *image* and *value*.

In this article, I am not going to make any objects of the *float_unit.class* just described. Instead, I will use *float_unit.class* as the parent for each units class like this:

```
with float_unit;
package hour is
   type class is new float_unit.class;
end hour;
```

These four lines of code create a new class package, *hour*, that is a subclass of the *float_unit* class package. As such, *hour* inherits all the operations that were defined in the original package. A happy feature of this approach is that you do not need to write a package body for *hour* because Ada attributes the body of *float_unit* to it. You could create and use the objects of *hour* as shown in Example 2, page 53.

The *with* clause makes the *hour* class package visible to the *time_card* procedure. The *use* clause allows you to use the addition operation without having to use functional notation. Without *use*, you would have to write

```
hours_worked := hour."+"(job_1,
                         job_2);
```

which is not as easy to read. The *hours_worked* declaration and the two lines following it create objects (variables) of that class. Each assignment statement (:=) gives an object a value.

So far, we have created one new package, *hour*, that is built up from the prototype dimensional unit package, *float_unit*. To create other dimensional units, you could repeat the four lines of code that you used for *hour*, and just change the package name to the name of the desired unit. But you can take advantage of that similarity to reduce the number of lines of code from four to two, by using an Ada feature called *generic*. A generic program unit in Ada is like a template for making similar program units. Oversimplifying a bit, you might think of generic as a macro capability. Here, then is the template:

Dimensional units arise in any application that has to deal with physical quantities

```
with float_unit;
package mile is new unit;

with float_unit;
with hour;
with mile;

package mile_per_hour is
     type class is new float_unit.class;

     function "/"(left : mile.class;
                  right: hour.class
                 ) return class;

end mile_per_hour;
```

Example 3: *Using the hour class and a new mile class to create the mile_per_hour class*

```
with hour;
with mile;
with quotient_unit;

package mile_per_hour is new quotient_unit(
     numerator_class => mile.class,
     denominator_class => hour.class);
```

Example 4: *Installing the specification and the body parts for the packages listed in Examples 2 and 3*

```
with unit;
package foot is new unit;

with foot;
with product_unit;
package square_foot is new product_unit(
     class_a => foot,
     class_b => foot);


with foot;
with square_foot;
with product_unit;
package cubic_foot is new product_unit(
     class_a => foot,
     class_b => square_foot);
```

Example 5: *Creating new composite units by applying an existing generic package as many times as necessary. In this case, a package for cubic feet is built in three steps*

```
with float_unit;
generic
package unit is
   type class is new float_unit.class;
end unit;
```

Now, you can create new units packages like this:

```
with unit;
package hour is new unit;
```

This new package called *hour* is identical to the one created earlier that used four lines of code. The new *hour* inherits all the operations of *float_unit*. The process of creating an actual package from a generic template is called *instantiation*.

Fundamental and Composite Units

In any application area, there will be "fundamental" units and "composite" units. For example, *mile* and *hour* might be fundamental units while *miles per hour* is a composite

```
with float_unit;
with hour;
with mile;
with mile_per_second;

package mile_per_hour is
     type class is new float_unit.class;

     function "/"(left  : mile.class;
                  right : hour.class
                 ) return class;


     function convert (mps :
                       mile_per_second.class
                      ) return class;
end mile_per_hour;
```

Example 6: *Converting routines to couple a package with other packages*

```
with mile_per_hour;
with mile_per_second;
package mph_mps_convert  is

     function relation(mph : mile_per_hour.class)
                       return mile_per_second.class;

     function relation(mps : mile_per_second.class)
                       return mile_per_hour.class;
end mph_mps_convert;
```

Example 7: *Modelling a relationship as an object*

unit, made up of the other two. You can create all the fundamental units as classes that directly inherit all the valid operations of the *float_unit* package. You can then create the composite units as classes with added or overridden operations. If you had an application dealing with velocity, you could create a *mile* class, reuse the *hour* class just described, and then create the *mile_per_hour* class as shown in Example 3 on page 54.

In Example 3, the *hour* class and the *mile* class inherit all the operations defined in the *float_unit* class. The *mile_per_hour* class inherits all those operations, but adds a new division operation. The new division operator allows you to divide a *mile* object by an *hour* object to obtain a *mile_per_hour* object. The *mile_per_hour* package requires its own body so that the details of the new division operation can be coded.

You can reduce the effort of writing composite unit packages by again using Ada's generic facility. The generic facility helps you create one package for composite units made by dividing two other units (such as miles/hour), and another package for units made by multiplying two other units (such as square feet or foot-pounds). The complete specifications and bodies for both packages are shown in Listings Two and Three, starting on page 94. You customize these for your particular case, as shown in Example 4 on page 54.

As you look at the generic packages in Listings Two and Three, notice that *quotient_unit* has a division operation as you would expect, but it also has two multiplication operations. Similarly, *product_unit* has two multiplication operations, but it also has two division operators. In both cases, the "extra" operations are ones that you would normally expect to perform on any composite unit. For example, the *mile_per_hour* class package created from *quotient_unit* provides the division operation so that you can divide *mile* objects to *hour* objects to *mile_per_hour* objects. But it also provides the multiplication operations so that you can multiply *mile_per_hour* by *hour* objects to obtain *mile* objects.

What about composite units that are square or cubic units? You would simply apply the *product_unit* generic package as many times as required; for a package for cubic feet, see Example 5, on page 54.

Conversions Between Units

Perhaps as you were reading, you wondered what to do about units that are simply scaled versions of each other. For example, how should you convert *mile_per_hour* to *mile_per_second*? Should each be a separate class? Should you write a function to convert one to the other? My answer is that they need to be separate classes because they are not directly combinable: you cannot add miles per hour and miles per second. But in simple cases, a special conversion function is not necessary. You could just multiply or divide by the proper constant to perform the conversion. For the case just described, the conversion factor is 1 hour/3600 seconds. Here is the code you might write:

```
with hour.class;
with second.class;
with mile_per_hour; use
                        mile_per_hour;
with mile_per_second; use
                    mile_per_second;
...
mph : mile_per_hour.class := 60.0;
mps : mile_per_second.class;
...
mps := mph * hour.class(1.0)/
  second.class(3600.0);
...
```

```
generic
     -- Import one kind of class:
     type class_a is digits <>;
     -- Import the other kind:
     type class_b is digits <>;
     -- Import the conversion factor
     a_to_b_factor : in float := 1.0;
package class_a_class_b_convert is

     function relation (a : class_a
                        ) return class_b;


     function relation (b : class_b
                        ) return class_a;

end class_a_class_b_convert;


package body class_a_class_b_convert is

     function relation (a : class_a
                        ) return class_b
     is
     begin
               return class_b(float(a) * a_to_b_factor);
     end relation;


     function relation (b : class_b
                        ) return class_a;
     is
     begin
               return class_a(float(b) / a_to_b_factor);
     end relation;

end class_a_class_b_convert;
```

Example 8: *Generalizing relationship objects for dimensional unit applications by creating a class that provides functions to go both ways. This is implemented as a generic package that imports the conversion factor and the two objects that are to be related.*

(The notation *hour.class (1.0)* is Ada's way of expressing the conversion of a floating-point number to the *hour.-class* type.) You have to be careful, though. The *mile_per_hour* package knows how to perform the multiplication by *hour* objects to produce *mile* objects, so that operation must occur first. Then the *mile_per_second* package knows how to divide *mile* objects by *second* objects, so that operation needs to occur next. Imagine that you put the operation in reverse order like this:

```
mps := mph / second.class(3600.0)
                        *hour.class(1.0);
```

The compiler would reject this line because the *mile_per_hour* class does not know anything about seconds.

Object Coupling

In general, it is probably safer and easier to create a function to perform the conversion from one unit to an -dother. But where should you put that function? In the mile/second example, should you include the conversion function in the *mile_per_hour* package or the *mile_per_second* package? Or in both? Or neither? If neither, then where would the function go? If you put the function in the *mile_per_hour* package, then that package is no longer self-contained and independent. The package then depends on, or is linked to, the *mile_per_second* package. You can see this if we try to put a conversion routine into that package, as shown in Example 6, page 56.

Notice that Example 6 does not use the generic unit package. This is so you can add the special divide and convert functions. More importantly, notice that the *convert* routine operates on a class of objects (*mile_per_second*.class) not defined in the same package. When this happens, the two classes or objects are said to be *coupled*.

You might ask why it is bad to couple *mile_per_hour* to *mile_per_second*, but not bad to couple *mile_per_hour* to *hour*, *mile*, and *float_unit*. The reason is that it is impossible to have *mile_per_hour* without *mile*, *hour*, and *float_unit*, but it is possible to have *mile_per_hour* without *mile_per_second*. If two classes or objects are coupled, a change in one will usually affect the other. This propagation of changes can complicate software maintenance. So you should strive to minimize coupling between object and classes. The minimum coupling is that which exists in the domain you are modeling. If coupling exists in the real (physical or mental) world, your software should have similar coupling, otherwise it would not be an accurate model of this application area. In general then, only introduce coupling that is warranted by the nature of the objects and classes you are modeling.

So, how can you minimize the coupling between these two unit classes? Remember that in object-oriented design, classes and objects are created based on physical or mental classes and objects, and that dimensional units are mental objects — namely, characteristics. What you have now is another mental object, called a *relationship* — specifically, the fact that one dimensional unit is related to another dimensional unit by some constant factor. This relationship is shown as an object in Example 7, page 56. (Only the package specification is shown in the example, you would still have to code its body.)

Here, an object called *mph_mps_convert* has been created that models the relationship between *mile_per_hour* and *mile_per_second*. This relationship works both ways. You can convert *mile_per_hour* to *mile_per_second* and vice versa. Thus, there are two functions, one for each direction. Notice that even though both functions have the same name (*relation*), the Ada compiler chooses the correct one based on the type of the function's argument when the function is invoked. This new object is coupled to the two packages that it ties together, but it leaves the packages independent of each other. If for some reason you need to make a change in the *mile_per_hour* package, it might affect *mph_mps_convert* but would not affect the *mile_per_second* package.

The package *mph_mps_convert* is not a class because it cannot create any subclasses or objects (it does not export a type and it is not generic). But it is a full-fledged object because it has publicly accessible procedures that operate on otherwise inaccessible data, the conversion constant. You also can call it an object because it is a model of a mental object.

The naming convention followed here is to construct the name of the package by concatenating the names of the two related classes with the name of the relationship. The package name then identifies the two objects and the specific relationship. The reason for following this convention is that any two classes may be related to each other in many ways and each relationship should have its own package with a unique name. The name "relation" chosen for the functions in the package is used simply to indicate that this is a relationship object.

You can generalize this relationship object for dimensional units into a class as Example 8 illustrates on page 59. The class is implemented as a generic package that imports the two objects that are to be related and the conversion factor that relates them. A generic package like this represents a class because it is used to create packages that represent objects.

Once the *class_a_class_b_convert* package of Example 8 exists, you can use it as shown in Example 9, next page, to make conversion objects for any two units classes that are related by a constant.

Example 10, next page, shows a sample code fragment that uses the *mph_mps_ convert* object package from Example 9 to convert from *miles-per-hour* to *miles-per-second*. You can see that the conversion amounts to no more than a subroutine call, but by careful design we have avoided unwanted coupling and solved the problem of "where to put the conversion routine."

(continued from page 61)

Conclusion

The advantages of using the method described here to implement dimensional units are that it enlists the language's type checking abilities to keep you from incorrectly using dimensional units, it makes the creation of new unit classes easy, it handles "composite" units in a consistent way, it uses relationship objects to avoid object coupling, and it makes all the units and relationships reusable.

The concepts and techniques presented in this article are useful in many application areas — not just dimensional units. I have focused on dimensional units here because they are abstract, and because newcomers to the OOD field have difficulty with abstractions when they try to apply object-oriented design to practical problems.

```
with mile_per_hour;
with mile_per_second;
with class_a_class_b_convert;
package mph_mps_convert is new class_a_class_b_convert(
     class_a => mile_per_hour.class,
     class_b => mile_per_second.class,
     a_to_b_factor => 3600.0);
```

Example 9: *Using the relationship class described in Example 8*

```
with mile_per_hour;          use mile_per_hour;
with mile_per_second;        use mile_per_second;
with mph_mps_convert;
...
mph : mile_per_hour.class := 60.0;
mps  : mile_per_second.class;
...
mps  := mph_mps_convert.relation(mph);
```

Example 10: *Code fragment showing the use of the mph_mps.convert object created in Example 9 to convert 60 mile_per_hour into mile_per_second*

Acknowledgments

Thanks to my coworkers at Motorola, Randy Facklam, Mike Purcell, and Carl McNealy for their questions, ideas, and suggestions on the concepts discussed in this article.

Notes

Do-While Jones "Dimensional Data Types." *Dr. Dobb's Journal*, May 1987, p. 50.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

The State-of-the-Art in Modula-2

The growing interest in Modula-2 has spawned a new crop of compilers for the PC. Here's how they stack up.

by Kent Porter

In the past couple of years, Modula-2 has progressed from the status of a newcomer to that of a viable language for both applications and systems software development. *DDJ* last compared Modula-2 compilers in the August 1986 issue. Since then, the state of the art has advanced dramatically, and so has interest in Modula-2 among *DDJ* readers. Recent introductions have still further strengthened Modula-2 as a potential successor to both Pascal and C. It's time to take Modula-2 seriously, and consequently this article surveys four leading Modula-2 development systems.

Three of these Modula-2 compilers have been introduced this year. The old-timer is Logitech, which is now in Version 3.0 and has been around for several years. The latecomers are FTL from Workman Associates, TopSpeed by Jensen and Partners International (JPI), and Stony Brook. Others are available as well, but review copies did not arrive in time to be included.

To one degree or another, all of these products provide an "environment"—a source program editor that interacts with the compiler—along with optional command-line compilation and widely varying program-development aids. Two compilers have their own linkers, and the others use the DOS linker. Without exception, they all support the core language as defined by Wirth, plus they offer extensions. And, of course, some claim to be the fastest, a clamor that will be dehyped with the benchmarks later in this article.

While support for the basic elements of Modula-2 effects a degree of standardization, it does not ensure portability of a Modula-2 source program from one implementation to another. Like C, Modula-2 is a limited language that derives much of its power and flexibility from external routines. For example, Wirth recommends that I/O service routines reside in certain libraries and take thus-and-such parameters, but he doesn't mandate them. Nor does any formal standard for Modula-2 yet exist. Consequently, implementors take liberties. These liberties inevitably lead to discrepancies that diminish portability.

Table 1, page 67, shows a sample of declaring the external routines *WriteString* and *WriteLn*, which write a string and a newline, respectively, to the display.

Logitech and Stony Brook implement Wirth's recommended InOut library, but FTL and TopSpeed do not. In fact, TopSpeed doesn't even use the same procedure names (although it does provide alternative libraries that do). Therefore, moving a program from one implementation to another is likely to necessitate making changes to accommodate the deviations from Wirth.

Deviations are not necessarily bad, and certainly not unique to Modula-2. If you buy one compiler and stick with it (as most programmers do), you'll never confront the issues of portability. Languages governed by formal standards also cause headaches in porting between compilers. Ada comes as close as any language to being universal, yet even it is imperfect.

Turning to specifics, this article profiles each of the products. Space is limited, so the focus is on the main features, which are summarized in Table 2, on page 67. The performance aspects are covered later in the section "Benchmarking the Compilers."

FTL

A few years ago, no one would have believed that $49.95 would ever buy so much program-development software. Workman Associates' FTL Modula-2 comes with an editor, compiler, symbolic debugger, and other utilities, plus a generous array of standard modules. Considering the quantity and quality of the software, FTL delivers the most per buck of the development systems reviewed here.

The editor uses the whole screen and operates with a Wordstar-like command structure. If you don't like Wordstar, or if you want to add custom commands, you can build

Kent Porter, DDJ's senior technical editor, also writes the "Structured Programming" column. Kent can be reached through CompuServe at 7670,51 or through MCI: kporter.

your own macros to do anything within reason. To do more than basic editing operations, the ^O command pops up an options menu that provides (among other things) interactive compiling, linking, and running of the program under development. The options menu also provides control over editing windows, access to the DOS shell, and other useful features. If you really want to change the editor, the source code—written in FTL Modula-2—is available at a modest extra cost.

Both the compiler and the proprietary linker are also available at the command-line level, and respond to a number of switches that influence the resulting .EXE file. For example, the command *ML myfile/C/V* invokes the linker and tells it to create a compact (64K) code segment and the minimal possible data segment. You can also enter switches from the editor's options menu.

FTL furnishes a symbolic, full-screen debugger with source tracking, breakpoints, trace-back, variable watching, and other features. Most operations are driven by function keys, and a pop-up window on the right side of the display provides reminders. A slick display option enables you to chase down a pointer to see the record it indicates, control the format of real numbers, and view portions of arrays. An associated execution profiler reports the total amount of time spent executing each statement in a program — useful for tuning an application.

Among the utilities are the MLU librarian (which enables you to create and manage libraries of .OBJ modules), a documenter that shows dependencies among modules, and a *make* utility. FTL also has a *TRIMMER* utility to remove unused hunks of code from the .EXE file.

FTL Modula-2 comes in two versions. The small model, which does programs up to 64K code and 64K data, sells for $49.95. The large model compiler, which is $30 more and the one tested here, handles programs of any size (subject of course to the usual segmentation rules). The large model includes optional 80x87 coprocessor support. Because of file-name conflicts, the systems—if you have both—must reside in different directories. Workman Associates sells the optional editor toolkit for $39.95. Additionally, FTL Modula-2 compilers are available for CP/M (Z80) and the Atari ST.

If you're looking for an inexpensive compiler to learn Modula-2 and develop some software, this is the one to buy. It gives you a lot for the money. If you also want high performance, you'd better check the benchmark report later in this article.

Logitech

The oldest and best-known of the Modula development systems, the Logitech compiler, is a modular offering comprised of a core package and an extremely fine accessory toolkit. This is consistent with the nature of the language it supports. The $99

Wirth:	FROM InOut IMPORT WriteString, WriteLn;
FTL:	FROM Terminal IMPORT WriteString, WriteLn;
Logitech:	FROM InOut IMPORT WriteString, WriteLn;
TopSpeed:	FROM IO IMPORT WrStr, WrLn;
Stony Brook:	FROM InOut IMPORT WriteString, WriteLn;

Table 1: *Incompatibilities in declaring common external I/O routines*

	FTL	Logitech	TopSpeed	Stony Brook
Editor:				
Style	Wordstar	Point	Turbo	Brief-like
Customizable	Macros	.INI files	.MNU file	No
Compiler:				
Switches	7	21	12	18
Directives	None	6	24	None
Memory models	2 [1]	1 [5]	Note [10]	6
Overlays	No	Yes	Yes [9]	No
Inline assembly	Yes	No	No	No
8087 support	Yes	Yes	Yes	Yes
'286 support	Yes[2]	Yes	No	No
Linker	Included	DOS[6]	Integrated	DOS
Switches	11[3]	N/A	7	N/A
Debugger	Symbolic	Note [7]	Symbolic[9]	Symbolic
Make	Yes	Optional	Integrated	Yes
Librarian	Yes	No	Integrated	No
Other utilities	Yes	Optional	No	No
Standard modules	28	63	13	23
Standard identifiers	233	479	259	157
System requirements:				
Lowest DOS version	2.0	2.0	2.0	2.0
Memory	256K	512	384K	?
Floppies	2	2	2	2
Hard disk	N/R	Recomm.	Recomm.	Recomm.
Color monitor	N/R	N/R	Recomm.	N/R
Base price	$49.95 [4]	$99.00 [8]	$99.95	?
Version tested	1.08	3.03	1.10	1.10

Notes:

1. FTL small and large models are separate products
2. In large model only, via a linker switch
3. In large model. Four for small model linker
4. Small model. Large model is $79.95
5. Large model only
6. Logitech linker available with optional toolkit (not evaluated here)
7. Base package includes post-mortem debugger. Interactive symbolic debugger available in optional toolkit
8. $249.00 with optional toolkit
9. Optional and extra
10. Programmer-defined memory models

Table 2: *Modula-2 compilers: features, requirements, and pricing*

core package consists of the POINT editor, compiler, post-mortem debugger, and Logitech's impressive libraries. For $249 you can get the compiler and the toolkit bundled together. This is well worth the extra cash for serious programmers.

Because this is the same Logitech of mouse fame, POINT is a heavily mouse-oriented, menu-driven editor. You can use it without the mouse, but you will not want to. POINT is infinitely reconfigurable via .INI files that use a quasi-programming language for specifying menus, actions to be taken, and such. The editor itself is friendly, fast, and capable of a wide array of word processor-like functions.

Of primary interest in program development is POINT's M2ASSIST pull-down menu, which has selections leading to the syntax checker, compiler, and linker (among other things). You can also run your program without leaving POINT.

Logitech includes two compiler versions called the fully-linked and overlay versions. The fully-linked version requires 512K; the overlay version requires 290K. The compile-time penalty for running the overlay version is in the range of 15 percent to 20 percent.

To save you the nuisance of typing command-line switches, the compiler looks to a text file called M2C.CFG. This file contains all the switches. To activate a particular switch, place a slash in front of it. Disabling the defaults for switches S, F, R, T, A, and O has a dramatic effect in reducing the code size and execution times of .EXE files. If you elect to leave the defaults active, the compiler inserts all types of run-time checks. It's a good idea to have the defaults on during development, then turn them off for the final compile in order to optimize size and speed.

The core package's *PMD* utility is a symbolic post-mortem debugger. If a program crashes, you can run PMD to find out the values of all variables, state information, the point of failure, and the procedure call chain. In order for PMD to work, the failing program must have imported Logitech's DebugPMD module, which activates on abnormal termination and writes a memory dump to disk.

dhrystone	A statistically balanced mix of operations
sieve	Array indexing and integer arithmetic
fib	Integer arithmetic and recursion
acker	Recursion and integer arithmetic
fpmath	Transcendental floating-point arithmetic
Without 80x87	FP emulation package efficiency
With 80x87	Efficiency of generated code using the 80287 math coprocessor
qsort	QuickSort algorithm
shsort	Shell-Metzner sort algorithm
cortn	Coroutine context switching speed*
ncortn	Same task as cortn, but without coroutines*

* Modula-2 supports coroutines: tasks that call each other as equals rather than as superior/subordinate. The difference between *cortn* and *ncortn* measures the time required for 100,000 coroutine context switches (see sidebar for more information).

Table 3: *Summary of benchmark programs and what they test*

The toolkit furnishes a more conventional interactive debugger, which enables you to watch a program run. The debugger subdivides the display into windows that contain the executing source code, the calling chain and breakpoints, watched variables, and a module list with an indicator to the current module. You can also open other windows to dump memory, view the application screen, and so forth. A number of commands provide user control over the session and the display.

The toolkit raises Logitech Modula-2 to the level of an advanced professional development system. In addition to the run-time debugger, it includes a proprietary linker, a *make* utility, a disassembler, a cross-reference program, and other useful utilities. It also provides source code for the libraries so that you can make modifications and enhancements, then simply recompile. Of the products reviewed, Logitech furnishes the most comprehensive set of tools.

The libraries are also extremely rich. They offer almost twice as many intrinsic procedures and identifiers (479) as any of the other Modula vendors. Logitech's longevity in this market is obvious, and they haven't neglected their responsibility to enhance the product. Looking at some of the calls, you might wonder how you'd ever use them. The point is that if you need them, they're there.

Logitech also deserves applause for its superb documentation. The package as a whole has four softcover manuals that are well-organized and indexed, clearly written, and replete with screen shots and other graphics. The 412-page compiler manual includes a lucid introduction to Modula for Pascal programmers.

Logitech has also recently begun shipping a $349 OS/2 version of the package. To date, this is the only Modula-2 compiler for OS/2.

JPI's TopSpeed Modula-2

Although new to the United States, TopSpeed from Jensen and Partners International (JPI) has been marketed in Europe for the past year under the name JPI Modula-2. Because it attracted favorable notice there, its reputation has preceded it on this side of the Atlantic. This recognition is well deserved, because TopSpeed is surely one of the finest new products introduced to date in the PC arena. If you need anything to convince you to move up to Modula-2, this is it.

The essence of TopSpeed's system is an integrated environment that bears a striking resemblance to that of the Turbo languages. Even many of the Alt-commands are the same. If you're familiar with Turbo Pascal, Turbo C, and so forth, you'll feel right at home from the start. And even if you're not comfortable at the beginning, you soon will be because the commands are intuitive. For example, Alt-C means compile, Alt-R means run after *make*, and so on. You can always back out of a misdirected command with Esc.

The Wordstar-based editor is reconfigurable by editing the M2.MNU file. This allows you to modify the default menu and add new commands (such as your own utilities) and integrate them into the user interface. You can work with as many as four configurable editing windows

at a time, with different files in each or concurrently updated instances of the same file in multiple windows.

Compilation is somewhat different than in the Turbo environments. The compiler doesn't stop on the first error. Instead, it goes all the way through and displays a red meter to show percentage completion. When it's done, the editor rests the cursor on the first offense. You can fix the error, then move to the next error with the F8 key. At each line containing an error, a red-backed diagnosis appears near the bottom of the display. When you've repaired the damage, you recompile with one of the Alt-commands.

The environment integrates the compiler and linker, and it's blazingly fast. If you resist environments (and I can't imagine why you would), you can use your own favorite editor and operate the compiler/linker on a command-line basis. The commands are as follows:

To compile:	M2 /C yourmod
To link:	M2 /L yourmod

TopSpeed deviates from the de facto Wirth standard in the naming of default I/O routines. For example, Wirth specifies the module InOut exporting *WriteString* to output a string and *WriteLn* for a newline. TopSpeed's implementation instead uses *WrStr* and *WrLn*, respectively, exported from a library called IO. This might make you uncomfortable, especially if you're just learning Modula-2. For that reason, one of the three distribution diskettes contains a directory called \CORE, which has a number of compatibility modules (one of which is InOut) that allow you to use Wirth's recommended libraries and standard procedure calls. All you have to do is compile them, and TopSpeed becomes compliant with the Wirth standard and with other Modula compilers.

Of the development systems reviewed here, only JPI's TopSpeed supports dot-addressable graphics with intrinsic calls such as Circle, Line, and Polygon. It's not yet at the level of Borland's .BGI drivers, but ahead of the other Modula products. TopSpeed also provides advanced text manipulation and cursor control.

An optional symbolic debugger comparable to CodeView is available from JPI, as is a $59.95 toolkit. This toolkit includes an assembler and ROM-burning software, plus modules for communications, terminate-and-stay-resident programs (TSR's), critical error handlers, EMS, and overlays. It also provides source code for TopSpeed's start-up and run-time libraries.

DDJ doesn't give unqualified raves very often, but there's no question about it in this case: JPI's TopSpeed Modula-2 is first-rate.

Stony Brook

As the benchmark discussion later reveals, Stony Brook delivers outstanding performance. Unfortunately, their tools don't measure up in comparison with the others. Stony Brook comes with an editor, compiler, libraries, and a rudimentary debugger. That's all.

The editor is somewhat like Brief, though less capable. For example, you can't customize it, and you can only work with two files at a time on a split screen. The *Alt-C* command enables you to compile the file in the active window from within the editor. An error positions the cursor on

A New Benchmark for Modula-2 Compilers

Among the mainstream languages, Modula-2 is the first to support concurrent processes through standard procedure calls. Most computers are single-thread machines, so concurrency is achieved by time-division multiplexing: one process runs for a time, then yields the machine to another process, which takes its turn and then reverts to the first, and so on until both complete. In Modula-2 terminology, such processes are called *coroutines*. Coroutines exist as equals, each periodically deferring to the other, which picks up where it most recently left off.

An issue with coroutines is *context switching* (the saving and restoring of the machine state during control transfers). This entails overhead, and the question becomes one of quantifying it. A benchmark is clearly needed for comparing the relative performance of competing compilers in handling coroutine switching. That's what *DDJ* presents here.

In fact, two benchmarks are needed that perform exactly the same task — one using coroutines and the other not. The Modula-2 programs CORTN.MOD and NCORTN.MOD (see Listings, page 100) both generate a 50,000-character string in lowercase, then shift that string to uppercase. The difference is that CORTN uses a coroutine to count the number of characters shifted, whereas NCORTN uses a normal procedure call. Granted that the task is trivial, but this is consistent with benchmarks in general. The objective is to measure the relative amount of overhead introduced by coroutines.

So here's what you do. Compile and run CORTN and NCORTN, timing the execution period for each as precisely as possible. Then calculate the overhead introduced by coroutine switching. If C is CORTN run time and N is that for NCORTN, the percent overhead (O) for coroutine switching is

$$O = (C - N) / C$$

Thus, if CORTN runs in 10.30 seconds and NCORTN runs in 1.68 seconds, the percent overhead for coroutines is 83.73 percent.

In comparing coroutine performance, the lower the overhead percentage, the more efficient a particular compiler is at handling coroutines. For example, in Table 4 accompanying this article, FTL has the most efficient coroutine handling because its percentage overhead is the lowest.

DDJ would like to place these Modula-2 benchmarks and the method for evaluating them in the public domain. If you have comments, please submit them in writing to *DDJ*, Attn. Kent Porter, at the address on the masthead, or to kporter on MCI or 76704,51 on CompuServe. The results will be published in a future issue. — KP

the offender and generates a message explaining why the compiler choked. It's usually sufficient to suggest a fix.

But editing and compiling are all you can do from within the editor. To link and run, you must exit to DOS. Although there's nothing wrong with this strategy, integration is not as good as with the other versions reviewed here.

The debugger is primitive: better than nothing, but not as good as it might be. There's nothing visual about it. You type commands and get answers on a blind line-by-line basis, without the benefit of watch windows and dancing execution bars showing where you are in the source. The only way to work with this debugger is with a line-numbered source listing by your side.

I like Stony Brook Modula-2, and I've said so in the "Structured Programming" column. Even its competitors make admiring noises about it. But performance, and not features, sell it. If Stony Brook can retain the performance while ramping up the features and libraries in future releases, they'll have a strong development system.

Benchmarking Compilers

Benchmarks are always controversial, of course, and these will probably be no exception. In evaluating these results, bear in mind that any given benchmark measures performance of a certain limited set of activities. Therefore, a benchmark provides only a general idea of how one computing element (a compiler in this case) stacks up against others of its kind. The set of benchmarks as a whole provides a reasonably good feel for overall relative performance, but the real test is your particular application on your machine.

These tests were all performed on a Tandon AT clone running at 8 MHz with no-wait-state memory and a 35 ms, 40-Mbyte hard disk. The timing figures were provided by a supervising program that ties into mode 2 of the 8253 timing chip, which provides a granularity of 838 nanoseconds. The timer program carries seconds out to four decimal places, which have been rounded to two places in the tables. The figures are the average of five executions each, and represent the total of load-and-run time.

In all cases, the compiler/linker command-line options were set to yield the best execution time for the resulting program. When possible, overhead operations (such as subscript range checking) were disabled. Where alternative libraries were available—as in the case of TopSpeed's limited run-time system and FTL's 8087-only floating-point, for example—I used them. Every effort was made to achieve the most favorable results.

The averages summarizing all except Table 5 are geometric, not arithmetic. A geometric average gives equal weight to all rows. A row that is much "bigger" or "smaller" than others doesn't dominate the results. For example:

	A	B
1	90.0	80.0
2	5.0	10.0
Total	95.0	90.0
Arith. avge.	47.5	45.0
Geom. avge.	36.8	52.6

Based on the arithmetic average, B seems to outperform A. Test 1 is inordinately long and Test 2 is very short. Giving equal weight to both tests by using a geometric average, you see that, in fact, A outperforms B.

The contenders were benchmarked with nine test programs shown in Listings One through Nine starting on page 100. If necessary, I changed library names in IMPORT statements and procedure calls to conform to the requirements of the tested implementation, all without altering the underlying algorithm. To the greatest extent possible, all programs are identical across all tested compilers.

FTL and TopSpeed provide their own linkers, which were used to

Program:		FTL	LogiTech	TopSpeed	Stony Brook
sieve	Compile	2.70	5.51	3.71	2.65
	Link	3.48	3.70	2.72	2.87
	Total	6.18	9.21	6.44	5.51
fib	Compile	2.53	5.07	3.66	2.65
	Link	3.57	3.70	2.72	2.86
	Total	6.10	8.77	6.38	5.51
acker	Compile	2.64	5.18	3.65	2.81
	Link	3.63	3.70	2.66	2.97
	Total	6.26	8.87	6.30	5.78
fpmath	Compile	3.76	10.72	5.94	4.62
	Link	15.24	8.25	4.45	5.17
	Total	19.01	18.97	10.39	9.79
qsort	Compile	2.79	5.84	4.15	3.13
	Link	3.62	3.69	2.84	3.30
	Total	6.41	9.53	6.99	6.43
shsort	Compile	2.79	6.15	4.44	3.19
	Link	3.67	3.68	2.96	3.29
	Total	6.46	9.83	7.40	6.47
cortn	Compile	3.45	5.44	4.34	2.91
	Link	8.50	3.57	3.30	3.35
	Total	11.95	9.00	7.64	6.26
ncortn	Compile	2.68	5.02	3.76	2.75
	Link	3.39	3.64	2.94	3.19
	Total	6.07	8.66	6.70	5.93
Geometric averages		8.20	10.37	7.48*	6.60

* Deduct 2.8 sec for the integrated environment

Table 4: *Modula-2 average compile and link times*

(continued from page 74)

convert .OBJ modules compiled by those products into .EXE files. Logitech and Stony Brook require the DOS linker (the version I used here was 5.01.20). The compile and link steps were performed separately on a command-line basis. I did not test the *make* utility for any of these compilers.

I used the Logitech translator to convert Dhrystone 1.1 from Pascal into Modula-2, and manually translated sieve, fib, and acker from C. The fpmath, qsort, and shsort programs were obtained from BIX and modified slightly, primarily to remove unnecessary output statements. The cortn and ncortn programs were written as new benchmarks specifically for Modula-2. These will be discussed later.

Table 3, on page 69, shows what the benchmark programs measure (in order of importance for each program).

Now for the results.

Compile and Link Times

Compile/link time is important because it affects the productivity of programmers; the less time from source to .EXE, the less idle time spent by the programmer.

For these tests, I dropped out of the integrated environment and measured time based on command-line mode. This is because accurate timings cannot be obtained within an environment. Actual execution times are shown in Table 4, on page 74, without the gap and typing time between compile and link. Systems with a *make* option automate the process and eliminate the inter-step gap, and so the totals for these systems are probably very close to the actual time.

The apparent winner here is Stony Brook, followed very closely by TopSpeed from JPI and then FTL. The spread is less than four seconds from first to third place. Logitech significantly trails in last place, yet its performance can hardly be characterized as bad.

Were it possible to accurately measure elapsed time within an environment, TopSpeed would undoubtedly emerge the winner. This is because the command *M2* loads the complete integrated environment, and then switches to batch mode based on the command line. To compile and link SIEVE.MOD, the commands are *M2 /C SIEVE* and *M2 /L SIEVE*. When the environment is already active, it's not necessary to reload *M2* from disk in order to do a link. Tests with the empty program

```
MODULE nothing;
BEGIN
END nothing.
```

revealed that the load time for TopSpeed is 2.8 seconds. Of those tested, TopSpeed is the only compiler with an integrated linker. Therefore, the compile/link totals should be adjusted downward by 2.8 seconds to estimate time within the environment, and TopSpeed emerges the clear winner at 4.68 seconds average.

Code Size

Table 5, below, shows the size in bytes of .EXE files flowing out of the compile/link process. Code size is not especially important for most applications. That is, a smaller .EXE is not necessarily better. It *is* important when the application is very large, or consists of many .EXE files that consume great amounts of disk space. For example, if a commercial program needs two distribution diskettes instead of one because of the summed .EXE file sizes, it becomes a cost factor. Otherwise, who cares?

Compilers tend to insert hunks of code as a matter of course. An example is a routine to initialize global variables. Such code cliches become fixed overhead in the .EXE file size. The NOTHING program cited previously yields the following .EXE file sizes in bytes:

FTL	3584
Logitech	3797
TopSpeed	609
Stony Brook	754

These represent the minimum size overhead of every .EXE file. If you want to determine the approximate amount of real working code included in a specific .EXE, subtract these fixed amounts.

If smaller isn't necessarily better, neither is larger (by definition) worse. As an example, note that Logitech

Program:		FTL	LogiTech	TopSpeed	Stony Brook
sieve		3584	3949	687	826
fib		3584	3891	683	818
acker		3584	3921	707	850
fpmath	(w/o '87)	25088	31225	13608	12138
	(with '87)	25088	30842	13608	12106
qsort		3584	4079	839	960
shsort		3584	4149	917	1002
cortn		13312	4498	2671	1504
ncortn		3584	3973	727	846
Geometric averages		10569	10598	2819	2793

Table 5: *Modula-2 benchmarks: code size*

Program:		FTL	LogiTech	TopSpeed	Stony Brook
sieve		8.61	8.31	3.33	3.85
fib		38.39	22.92	22.83	24.28
acker		20.71	11.82	10.19	12.14
fpmath	(w/o '87)	255.15	51.86	71.21	49.05
	(with '87)	12.08	45.28	10.30	11.63
qsort		7.85	6.00	4.04	3.62
shsort		11.69	11.53	8.98	10.38
cortn		5.59	6.22	10.30	12.85
ncortn		2.29	2.32	1.68	1.48
	overhead	3.30	3.90	8.62	11.37
	% o'head	58.97	62.68	83.73	88.48
Geometric averages		28.99	24.62	17.20	18.16

Table 6: *Modula-2 benchmarks: average run times*

yields an enormous .EXE file for fpmath, yet delivers second-best runtime performance in Table 6, page 76, while FTL produces a smaller file size and enormously slower execution for the same program. Performance doesn't have to do with file size, but instead with code efficiency. That's why file size is relatively unimportant.

Run Times

Table 5 lists the run times for the benchmarks. This is where "the rubber meets the road." All programmers want to produce fast applications, and they will sacrifice a certain amount of productivity and file size to do so. There is no equivocating here; either the resulting application is fast, or it's not.

Overall, JPI's TopSpeed Modula-2 lives up to its name, followed closely by Stony Brook. Logitech comes in a distant third, with FTL biting the dust.

What hurts FTL is its disastrously slow floating-point emulation. TopSpeed—out in front almost everywhere else—is third here, but still 3.58 times faster than FTL. If you remove the fpmath tests, the geometric averages are less widely spread, but still in the same order:

TopSpeed	8.67
Stony Brook	9.46
Logitech	10.80
FTL	13.11

These represent the performance rankings of the compilers' output .EXE files, and they're confirmed by the Dhrystone results.

But before leaving run times, compare the effects of adding an 80x87 math coprocessor to the hardware platform in the fpmath test. To perform these tests, it was necessary to recompile and relink fpmath with each compiler. The 80x87 has a negligible effect for Logitech, but for FTL the time plummets from 255 seconds to a mere 12, which is only a whisker behind first-place TopSpeed. Clearly, all but Logitech have expended great effort in exploiting the '87. The message here is if you're crunching a lot of floating-point numbers, buy an '87 and don't use Logitech.

Dhrystone

An industry-standard benchmark, Dhrystone is generally regarded as the most objective single predictor of compiled program performance. It's a synthetic application containing a statistically balanced mix of operations characteristic of the "typical" systems program. Dhrystone was devised by studying hundreds of non-floating-point programs, then constructing a 100-statement algorithm containing the following:

53 assignments
32 control statements
15 function and procedure calls

The entire program loops 50,000 times. Each iteration is one Dhrystone unit. Therefore, 50,000 divided by elapsed time yields the number of Dhrystones per second. The higher the number, the better.

Table 7, this page, shows Dhrystone results for the compilers. Stony Brook has the fastest raw compile/link time at 8.89 seconds. If you adjust TopSpeed's performance downward by 2.8 seconds to estimate environment time as discussed previously, TopSpeed becomes the fastest at 8.4. TopSpeed also wins by a wide margin in terms of .EXE size and Dhrystones per second. Logitech and FTL share last place, with Logitech leading by a hair. Stony Brook is exactly halfway between first and last place.

	FTL	LogiTech	TopSpeed	Stony Brook
Compile	4.62	12.26	7.17	5.11
Link	7.37	4.84	4.04	3.79
Total	11.99	17.10	11.20*	8.89
Size	9728	8957	2827	5638
Seconds	40.43	40.60	31.77	35.70
Dhrystones/sec	1237	1232	1574	1401

* Est. 8.4 sec in the integrated environment

Table 7: *Modula-2 Dhrystone results*

Conclusions

JPI's TopSpeed emerges as the clear winner. Its performance is spectacular, and it offers a powerful integrated environment similar to the Turbo languages, plus an optional symbolic debugger and toolkit. TopSpeed earns a standing ovation.

Stony Brook places a respectable second in all performance categories, running overall neck-in-neck with TopSpeed. The editor, debugger, and *make* utility form a useful, but rudimentary, toolkit for program development.

If you're a tools junkie, you'll love Logitech. Its optional toolkit furnishes oodles of them, and they're good. So is the performance of its output .EXE files—provided you turn off the default switches in the M2C.CFG file. Logitech's compile/link performance is lackluster, though, and its .EXE files are the largest of the crop. Nevertheless, the average execution time is fairly close to that of the leaders. The grand old man of Modula-2 compilers for the PC is still a formidable contender.

Although FTL ranks third in compile/link time and code size, its average .EXE run time consigns it to last place in performance. The only area where it really shines is in coroutine switching. What hurts FTL most is inefficient floating-point emulation. Overall, it's a serviceable compiler with an excellent set of tools—a genuine bargain at only $49.95.

So there you have the current state of the art in PC-based Modula-2 development systems, and the state of the art is very good indeed.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Arguments and Automatic Variables in Assembly Language

Interfacing assembly-language routines with high-level languages

by Raymond Moon

Although it's often useful and even necessary to write assembly-language subroutines that are called by a high-level language, a problem with Microsoft MASM is that it cannot use variable names to reference two important data classes: passed arguments and automatic (or local) variables. This shortcoming prompted me to develop a solution. This article describes techniques that allow the addressing of passed arguments and automatic variables using variable names.

In assembly language a variable is normally placed in a data segment, and the assembler associates a fixed offset with the variable name. Because passed arguments and automatic variables are created at run time by allocating space for them on the stack, such variables do not have fixed addresses — they can vary each time the same procedure is called. More significantly, if the function is recursive (that is, if the function can call itself), there might be multiple copies of these variables on the stack at the same time. One characteristic of these variables is that they be fixed in their relative position to each other; this is the key that allows the use of variable names to address them. The technique is to create a template using an assembly-language structure declaration to define the relative position of the variables.

Unfortunately, the method of passing parameters to subroutines differs among languages, among different compilers for the same language, and even among different versions of the same compiler. The code in this article was assembled using the Microsoft Macro Assembler, Version 5.0, and was designed to link with Microsoft C, Version 5.0. Also, unless otherwise stated, the discussion and code assume the small-memory model, which means that all procedures use *near* calls and all pointers are 16 bits long. Later I'll mention how to use this technique with all memory models and with different compilers.

Raymond Moon is currently writing the assembly-language procedures for a new remote bulletin-board system under development by the Annapolis Software Guild. He can be reached at 16005 Pointer Ridge Dr., Bowie, MD 20716.

Passed Arguments

To see how the structure declaration forms a template for addressing arguments, it's necessary to understand the arrangement of the stack upon entry to a called procedure. The standard C calling convention is to push arguments onto the stack in right-to-left order as they appear in the function argument list.

Using the C library function *strncpy( )* as an example (see Example 1, next page), the argument *n* is pushed first, followed by the pointer *string2* and finally the pointer *string1*. Last, the *near* call pushes the value of the instruction pointer (*ip*) register onto the stack to serve as the return address. The entry code of the called procedure must initialize a register to point to these parameters on the stack. The register used for this purpose is the base pointer (*bp*) register. The calling procedure's *bp* register value must be saved and then restored before returning to the caller. The standard entry code to perform this task is:

```
push bp
mov bp,sp
```

Now all the passed arguments can be addressed as positive offsets from the *bp* register. The stack at this point is shown in Table 1, on page 83.

The passed parameters can be accessed by using their relative offset from the *bp* register. Thus the code required to initialize registers with the passed parameters for *strncpy( )* could be:

```
mov di,[bp+4]  ; di points to
               ; destination string
```

```
mov si,[bp+6]  ; si points to source
               ; string
mov cx,[bp+8]  ; cx = number of
               ; char to copy
```

The trouble with using the relative address directly is that the code is hard to read and maintain. You can easily forget the offset to any passed argument, especially when reopening a program months after you originally wrote it.

A common method used to address passed arguments by name is to use the *equ* directive. For the *strncpy( )* example, these equates might be:

```
n          equ [bp+8]
string1    equ [bp+6]
string2    equ [bp+4]
```

The code to load the pointers and oucnt into the same registers would then be:

```
mov        di,string1
mov        si,string2
mov        cx,n
```

Although this improves code readability, it improves maintainability only marginally. If you change the calling argument list, you must recalculate the offsets and change the equates, which increases the possibility of error.

A better way is to let the macro assembler do it for you, which you can do by defining a structure for the part of the stack in which the passed parameters reside. For the previous example, the structure is shown in Example 2, this page.

A convenient result of the C calling convention (pushing the arguments onto the stack in reverse order) is that the order of the arguments in the structure is the same as in the function call argument list.

For C programmers, an easy way to use the offsets associated with structure names is to use the . (period) structure operator, which is similar to the . (period) operator in C. Using the *PARMS* structure, the earlier code fragment would be:

```
mov        di,[bp].string1
mov        si,[bp].string2
mov        cx,[bp].n
```

Although this is a little more complex because it uses structure notation, the code clearly identifies the variables as passed arguments.

Using structures in this manner increases code maintainability and versatility. If the argument list is changed, only the structure definition needs to be rewritten because the macro assembler automatically calculates the new offsets and uses them throughout the rest of the code. Another advantage is the ease with which the structure can be modified to work with the doubleword pointers required when using the compact-, large-, and huge-memory models, in which the data is not restricted to one 64K segment. You need only change the *dw* directive to *dd*, and the macro assembler will calculate the proper offsets. Table 2, page 83, shows the correlation between the C and assembly-language data types.

Automatic Variables

The real power of using assembly-language structures comes with the ability to create and address automatic variables. Storage for this variable class is allocated at run time from space available on the stack. Automatic variables allow assembly procedures to be recursive — that is, to call themselves without overwriting variables previously saved.

Automatic variables reside below the saved copy of the caller's *bp* register; remember that the stack grows downward as values are pushed onto it. Using the standard entry logic shown earlier, Table 3, page 84, shows a stack frame containing three integer-size arguments and three integer-size automatic variables.

The immediate problem shown in Table 3 is that the offsets to the automatic variables are negative with

```
char *strncpy(string1, string2, n);
char *string1;    pointer to the first string
char *string2;    pointer to the second string
unsigned int n;   number of characters to be
                  copied
```

Example 1: *Definition of strncpy( )*

Current Value	Stack
n	[bp+8]
string2	[bp+6]
string1	[bp+4]
ip register	[bp+2]
sp,bp =>saved bp reg	[bp]

Table 1: *The program stack after standard C entry logic*

```
PARMS struc       ; Passed arg addressing struc
      dw   2 dup(?) ; Saved ip and bp registers
   string1 dw    ?  ; Pointer to 1st string
   string2 dw    ?  ; Pointer to 2nd string
   N       dw    ?  ; # char to copy
   PARMS ends
```

Example 2: *Past argument structure*

C	ASM	C	ASM
[1]char	dw	Pointers:	
short	dw	small/compact	dw
int	dw	medium/large/huge	dd
long	dd		

[1]*Char* is expanded into a word before being pushed onto the stack

Table 2: *Correlation between C and assembly data types*

(continued from page 83)

respect to the value in the *bp* register but structure notation generates only positive offsets. The solution is to make the last variable the new addressing base. All automatic variables can then be addressed as positive offsets from this variable. Example 3, page 86, illustrates a structure for automatic variables using this method. The code required to make room for the automatic variables at run time and the equate required to define their addressing base is shown in Example 4, page 86.

This entry code should be used whenever automatic variables are used, even if there are no passed parameters to the procedure.

Example 3 shows some other advantages of using a structure for automatic variables. The assembler can calculate the exact number of bytes subtracted from the *sp* register using the *size* operator. Modifications are thus as easy as adding, changing, or deleting variables in the structure definition.

In creating such structures, it's important to ensure that all word and doubleword variables begin on a word boundary. You can do this by moving all byte-length variables to the bottom of the structure. If there is an odd number of byte-length variables, add an unnamed byte variable at the end so that the length of the entire structure is an even number. These precautions will ensure efficient memory accesses with 16-bit processors. If the code is to execute on an 80386, the only additional precautions are that any doubleword pointers should begin on a doubleword boundary and that the length of the *PARMS* structure should be divisible by 4.

The addressing of automatic variables using the equate shown in Example 3 would be:

```
mov ax, AUTO.INT__1
                        ; AX = INT__1
```

Again, the operator is used with all automatic variables. The assembler calculates the offset to this particular automatic variable as the sum of the negative offset from the *AUTO* equate and the positive offset generated by the variable's position in the structure. Although this sounds complicated, the assembler can handle it easily. Also notice that the use of *AUTO* clearly identifies *INT__1* as an automatic variable as opposed to a passed argument or a regular variable declared in the *DATA* segment. This easy recognition increases a program's readability and maintainability.

The code for returning from an assembly-language subroutine is simple:

```
mov     sp,bp
pop     bp
ret
```

The *mov* instruction clears the stack of all automatic variables and can be omitted if automatic variables are not used. The *pop* instruction restores the calling procedure's frame pointer.

A caution is in order here. The typing of variable names in assembly structures is not as strong as in C. This means that all variable names must be unique within a given assembly-language source file. The same variable name cannot be duplicated in two different structures or in a structure and the data segment because the assembler will not know which offset is associated with that name.

A Complete Example

Let's look at a function that ties together both passed variables and automatic variables. This function is called *file__length( )*, and its C definition is:

```
long file__length (filename, attribute)
        /* length of the file in bytes or
           -1L if the file is not found. */
char *filename; /* pointer to filename */
unsigned int attribute; /* file attribute */
```

Current Value	Stack
PARM__1	[bp+8]
PARM__2	[bp+6]
PARM__3	[bp+4]
saved *ip reg*	[bp+2]
saved bp reg	[bp]
INT__1	[bp–2]
INT__2	[bp–4]
INT__3	[bp–6]

Table 3: *Example of a stack with both passed variables and automatic variables*

ASSEMBLY LANGUAGE
(continued from page 84)

Using the *#defines* from Example 5, page 87, you might call this function as follows:

```
length = file_length('filename.ext',
                     NORM);
```

This call will search the default directory for "*filename.ext*" using the normal file attribute, which means that read-only, hidden, or system file attribute is set, it will not be found. If "*filename.ext*" is not found, the function returns *–1L*, otherwise, it returns the file size in bytes as a long integer.

The *file_length( )* subroutine uses DOS function *4eh* (Find First) to obtain information about the file. It extracts the length from this information and returns it to the calling procedure. Listing One, page 110, contains the assembly code for *file_length( )*.

```
_AUTO    struc    ; Auto var addressing struc
INT_1    dw   ?   ; 1st automatic variable
INT_2    dw   ?   ; 2nd automatic variable
INT_3    dw   ?   ; 3rd automatic variable
_AUTO ends
```

Example 3: *Automatic variable structure*

```
push bp          ; Save the called bp
mov  bp,sp       ; Initialize new frame pointer
sub  sp,size _AUTO
                 ; Make room for auto vars
AUTO equ [bp - size _AUTO]
                 ; Auto var addressing base
```

Example 4: *Standard entry code*

Lines 72 through 84 are required so that this assembly function can link to a Microsoft C program by defining the segments used by Microsoft C. If you use a different compiler, the documentation will probably contain all the information required to rewrite this section.

Lines 96 through 121 contain the structures required for this procedure. The first structure is for addressing the passed parameters, and it follows the guidelines discussed previously. The second structure defines the automatic variables. In designing this function, I wanted to minimize any interference with the calling program. Because DOS function *4eh* uses the disk transfer area (DTA) to pass the file information back to the program, my procedure needed to create a new DTA so that the contents of the current DTA would not be destroyed. Therefore, a new DTA is created as an automatic variable. The address of the old DTA must be saved here also so that the old DTA can be restored before the procedure returns.

The last two structures, lines 128 to 136, are required to address doubleword pointers and long integers. Because the assembler has a limited ability to address doubleword variables, these structures make the variables more naturally addressable. Although the same structure could be used for both, I used different ones so that the names more closely describe the actual actions being taken.

Now that the supporting structures have been defined and the required segments created, the actual code begins. *file_length( )* starts with the standard entry code. The *ds* and *es* segment registers are saved by pushing them onto the stack because they are changed by *file_length( )*. Notice that these are pushed onto the stack after the automatic variables have been created so as not to affect the automatic variable structure. Microsoft C uses *si* and *di* for register variables. If your routine uses *si* and/or *di*, you should save their values on the stack at entry and restore them before returning. (Because *file_length( )* doesn't use *si* and *di*, I didn't need to do this.)

From this point on, the code is fairly straightforward. The address of the current DTA is determined and saved, and a new DTA is created on the stack. The registers are then set up for the DOS *4eh* function call. If the carry flag is set upon return from DOS, the file was not found, so *–1L* is moved into *F_LENGTH*; otherwise, *F_LENGTH* contains the file length in bytes. Finally the old DTA is restored, and the length is loaded into the *dx:ax* registers in preparation for returning.

Table 4, below, summarizes the changes in the passed argument structure for different memory models. Although this table reflects Microsoft C memory conventions, it should work with most other compilers and languages.

```
#define NORM     0x00    /* Normal file     */
#define RO       0x01    /* Read-only file  */
#define HID      0x02    /* Hidden file     */
#define SYS      0x04    /* System file     */
#define VOL_ID   0x08    /* Volume ID       */
#define SUBDIR   0x10    /* Subdirectory    */
#define ARCH     0x20    /* Archive file    */
```

Example 5: *#defines for file attributes to be used with file_length( )*

Memory Model	Code Size	Data Size	Dup Operator Parameter	Pointer Size
Small	<64K	<64K	2	dw
Compact	<64K	>64K	2	dd
Medium	>64K	<64K	3	dw
Large	>64K	>64K	3	dd

Table 4: *Memory model effects on the PARMS structure*

(continued from page 87)

Conclusion

It's possible to increase the readability and maintainability of assembly-language programs by using named variables for passed parameters and automatic variables. The feature of the macro assembler that allows this is the structure definition that creates a template for the stack on which these two types of variables reside. Using the structure also simplifies the creation of automatic variables on the stack. Finally, using the structure declaration and the size operator, programmer math errors become a thing of the past.

I wish to thank Vince Castelli for his guidance while writing this article.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Microsoft Mixed-Language Macros

The Microsoft Macro Assembler Version 5.0 furnishes tools for mixed-language programming. Among them is a macro include file called mixed.inc that facilitates the writing of assembly-language procedures that link with Microsoft Basic, C, Fortran, and Pascal. The basic macros defined are:

- *setModel*—Sets the memory model from a text equate.
- *hProc <name [NEAR¦ğæţ,>"< USES reglist>] [,arg[:type]]...*—Starts a procedure with the option of saving reglist and with optional passed arguments.
- *hLocal var[:type] [,var[:type]]...*—Defines automatic variables.
- *hRet*—Restores the stack and returns.
- *hEndp*—Ends the current procedure.
- *ifFP* statement—Conditional assembly based on memory model.
- *FPoperand*—Provides the *es* segment override for data in medium, large, and huge memory models.
- *pLes register, pointer*—Loads a pointer into *es* and/or another register depending upon the memory model.
- *pLds register, pointer*—Similar to *pLes*, but loads *ds*.

These macros were designed around the Microsoft memory segment names, and they greatly increase the ease of writing assembly-language code. The *hLocal* macro has a couple of limitations, however. First, it is unable to handle automatic variables that are arrays; *hLocal* deals only with single objects, so it could not be used to simplify *file_length()* because of the two arrays used. Remember, a string is an array of type *char*. Second, any automatic variable declared as a byte is allocated word-length storage. Presumably this ensures word-length boundaries for faster memory access with 16-bit processors. Unfortunately, if true byte alignment is needed, as in the *file_length()* function, you cannot use *hLocal*.

Also, there is one potentially inconvenient result of using the *hProc* and *hLocal* macros: the entire macro invocation must be on a single source line. If the passed parameter or automatic variable list is long, the code sticks out far to the right, which can cause problems when printing the source listing. All in all, this is a small penalty.

The heart of the macro library is the *hProc* and *hLocal* macros, and each needs explanation. The *hProc* for *file_length()* could be:

```
cLang = 1
hProc FILE_LENGTH, <USES
        ds,es>, FILENAME:ptr, ATTR
```

The *cLang* equate indicates which calling convention is to be used. If *cLang* is defined as shown above, the C language calling convention is used. If *cLang* is undefined, the Fortran/Pascal/Basic calling convention is used instead. Assuming a small-memory model, a *near* procedure is created and two equates are generated with the following meanings:

```
FILENAME        equ [bp + 6]
ATTR            equ [bp + 8]
```

The Microsoft documentation warns that using these macros can slow down assembly. The reason is that *hProc* invokes 19 other macro calls. All these macro calls, however, make *hProc* quite capable. As mentioned earlier, it can handle C or Fortran calling conventions. The type *ptr* generates word-length pointers for small- and compact-memory models and doubleword-length pointers for medium-, large-, and huge-memory models. Timing the assembly of a small test file where the source and assembler were in a RAM disk, the increase was from two to four seconds. The source had two passed arguments and one automatic variable. Increasing the automatic variables to four added another second to assembly time. Although in relative terms this is a significant increase, in absolute terms it's scarcely noticeable and should not be a deterrent.

As already mentioned, I could not use the *hLocal* macro for *file_length()*. I looked to see if this macro could be easily modified to add the ability to handle arrays and byte-length variables. The faint of heart should not try to modify this macro, as parsing a line with macros is bewilderingly complex. As I could not see how to do it, these two limitations remain.

Finally, even if you do not use the other macros from *mixed.inc*, you should use *pLds* and *pLes*. These two macros greatly facilitate writing assembly-language code that will assemble as written in any memory model. One line of code will handle both word- and doubleword-length pointers as required for the memory model in use. — R.M.

XCMD and XFCN: HyperCard's Software Slots

HyperCard can be linked with more conventional languages including C, Pascal, Lisp, and assembly — if you know how

by Stan Krute

XCMDs and XFCNs are code resources that act as software slots for HyperCard. You can use them to create efficient extensions to the HyperTalk language. When HyperTalk can't (or won't) perform as you desire, XCMDs and XFCNs give you a powerful out. For instance, if HyperTalk lacks a particular command, you can write the necessary code in C, Pascal, or assembly; transform the code into an XCMD or XFCN; append the XCMD or XFCN to a stack or to HyperCard itself; then use the new command as if it were native HyperTalk.

Both XCMD and XFCN code resources begin execution at the first byte. When HyperCard transfers control to an XCMD or XFCN, it passes a single parameter via the stack: a pointer to an XCmdBlock data structure (see Example 1, page 91, for a C rendition). When control returns to HyperCard, it pays attention to a field in the XCmdBlock where XFCN's are supposed to place a handle to a return value, and ignores the same field for XCMDs. So one is a function, and the other is a procedure, as Pascalians would say. In HyperTalk, the analogous distinction is between function handlers and on handlers.

So far, most of the material from Apple and APDA that documents XFCN's and XCMDs has had examples written in C and Pascal. This article shows you the details of implementing an XFCN in 680x0 assembly language. The development environment is Apple's Macintosh Programmer's Workshop (MPW), Version 2.02. Listing One, page 108, shows the file XCMDandXFCN.a. It contains MPW assembler definitions of an XCmdBlock. I put the file in the MPW directory AIncludes. The file uses the MPW RECORD/ENDR pseudo-ops, which lets you define assembly language equates in ways quite analogous to C or Pascal. If you create such a file, and want to put it somewhere other than AIncludes, be sure to augment the AIncludes equate in the MPW Startup file.

My XFCN example (see Listing Two, page 108) is algorithmically trivial, simply implementing the useful C library function isAlpha, which controls the MPW build process. A few notes on the *make* listing in Listing Two: lines beginning with # are comments; the *f* symbol (Macintosh character code $c4) can be read as "depends upon" and the dependencies it indicates guide the *make* operation. The MPW tools mentioned are Asm, the MPW 680x0 assembler, and Link, the linker. Command lines were put together with *Commando*, the MPW CCLC generator.

The assembler takes the source code file, isAlpha.a (see Listing Three, page 108), and produces isAlpha.a.o, an object code file, and isAlpha.a.lst, a listing of the assembly. The linker takes the object file and produces isAlpha, a resource file containing my code compiled into an XFCN resource. I've set isAlpha up as a ResEdit document, with appropriate type and creator longwords and an icon, but that's just one of my completeness *shticks* and not necessary to the project.

Ignoring the source code's algorithm (a simple bounds test), the important thing to note about Listing Three is how I use the XCmdBlock

Stan Krute has been a Macintosh software developer since 1984. He is currently commuting to Redmond WA where he is writing programmers' documentation for the MS OS/2 Presentation Manager. He can be reached on CompuServe (User ID# 73137, 2121), or at 18617 Camp Creek Rd., Hornbrook, CA 96044.

to get information to and from HyperCard. Note how transparent the MPW record pseudo-ops used in XCMDandXFCN.a make my XCmdBlock interactions. Results sent back to HyperCard are in real live 0-terminated string format, stored in a memory block, and indicated via a handle. HyperCard takes care of memory block disposal details.

After the *make* procedure is completed, the code resides in isAlpha as an XFCN. You can then paste it into a stack using ResEdit or MPW. Example 2, below, shows the file isAlpha.r, which can be fed into the MPW resource compiler Rez. Example 3 and 4, below, show MPW make files that grab a HyperCard stack and then use *Rez* to append the isAlpha XFCN. The code in Example 3 will append the stack hctest, which is just a little XFCN to the one-button test stack that lives in the isAlpha folder. The code in Example 4 does the appendation for a stack outside the local world. As with Listing Two, the comments should make the cryptic commands understandable.

That's it. Linkup accomplished. You can now augment HyperCard with any kind of code you like such as the standard C and Lisp library functions I've recently added.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

```
typedef struct XCmdBlock {
     short paramCount;        /* # of entry parameters                        */
     Handle params[16];       /* array of handles to entry parameters         */
     Handle returnValue;      /* handle to a return value                     */
     Boolean    passFlag;     /* boolean: if true, HC passes message on       */

     void (*entryPoint) ();   /* pointer to HCard callback function (cbf)     */
     short request;           /* a cbf command code                           */
     short result;            /* HC's answer to a cbf command                 */
     long inArgs[8];          /* array of arguments in to a cbf command       */
     long outArgs[4];         /* array of arguments out from a cbf command    */
     } XCmdBlock, *XCmdBlockPtr;
# make instructions for testing the isAlpha XFCN via hctest
```

Example 1: *C rendition of an XCmdBlock data structure*

```
include "isAlpha" 'XFCN' (990) ;
```

Example 2: *The file alpha.r is used by the MPW tool Rez to append the isAlpha XFCN to a HyperCard stack*

```
# make instructions for testing the isAlpha XFCN via hctest

#the hypercard test stack depends on the resource file and the rez file
hctest     f     isAlpha     isAlpha.r
# to create the hypercard test stack, merge it with the XFCN
      # the rez file is isAlpha.r
      # the output file is the hypercard stack hctest
      # the XFCN will be merged into the existing stack
      Rez isAlpha.r -o hctest -a
```

Example 3: *The file hctest.make guides MPW as it appends an XFCN to a test stack located within a default directory*

```
# make instructions for adding the isAlpha XFCN to Zippy:hypercard:people

#the stack depends on the resource file and the rez file
people     f     isAlpha     isAlpha.r
# to create the people stack, merge it with the XFCN
      # the rez file is isAlpha.r
      # the output file is the hypercard stack
      # the XFCN will be merged into the existing stack
      Rez isAlpha.r -o Zippy:hypercard:people -a
```

Example 4: *The file people.make guides MPW as it appends an XFCN to a stack located within an arbitrary directory*

Listing One *(Text begins on page 18.)*

```
{ Read each line of an action chart file and filter out the bracket }
{ charactors.  Format the remaining text with regards to the left }
{ margin and indentation. }
{ NOTE - comments ending with a ** denote areas of this program which were }
{        left incomplete for the sake of brevity. }
{ You should customize the following constants to suit your particular }
{ tastes as far as indentation.  You may wish to use variables instead }
{ and obtain their values from command line switches or a configuration }
{ file. }

const

  margin_spaces    : integer = 0;         { customize these variables per }
  margin_tabs      : integer = 0;         { your particular taste. }
  spaces_per_level : integer = 2;         { Better yet, prompt for them at }
  skip_null_lines  : boolean = false;     { run time or read from a

{ These character sets are the block graphics characters which }
{ this program filters out. }

  top_set  : set of char = [' ',' '];    { alt-218, alt-213 }
  bot_set  : set of char = [' ',' '];    { alt-192, alt-212 }
  else_set : set of char = [' '];        { alt-195 }
  pass_set : set of char = [' ',' ',' ',' '];

                                         { space, alt-196, alt-205, alt-179 }
type
  str_type = string[80];
  chset = set of char;

var
  source,dest       : text;
  total_spaces,c_index,
  indent_adj,indent_level : integer;
  indent_str,work_str : str_type;


procedure advance_index(w_str : str_type; var ix : integer; test_set : chset);

{ accept a string, an index and a set of characters to the string test for. }
{ advance the index past all charactors in the specified set, returning an }
{ updated value of the index to the calling program. }

var
  w_str_len : integer;

begin
w_str_len := ord(w_str[0]);
while(true) do begin
  if ix > w_str_len then exit;
  if not (w_str[ix] in test_set) then exit;
  ix := ix + 1
  end
end;

begin  { MAIN PROGRAM LOGIC}

indent_level := 0;

{ validate command line parameters and manage file open errors ** }

assign(source,paramstr(1));
reset(source);
assign(dest,paramstr(2));
rewrite(dest);

{ read each line from the source file and filter out the characters which }
{ are used to produce the action chart brackets.  Write the resulting line }
{ to the destination file. }

while(not eof(source)) do begin
  readln(source,work_str);
  c_index := 1;
  indent_adj := 0;
  advance_index(work_str,c_index,pass_set);
  if c_index <= ord(work_str[0]) then
    if work_str[c_index] in top_set + bot_set + else_set then begin
      if work_str[c_index] in top_set then indent_adj := 1
      else
        if work_str[c_index] in bot_set then
          indent_level := indent_level - 1
        else begin      { must be in the else set }
        indent_level := indent_level - 1;
        indent_adj := 1
        end;
    advance_index(work_str,c_index,top_set + bot_set + else_set + pass_set);
    end;
  if c_index <= ord(work_str[0]) then begin
    total_spaces := margin_spaces + (indent_level * spaces_per_level);
    indent_str[0] := chr(margin_tabs + total_spaces);
    fillchar(indent_str[1],margin_tabs,#9);
    fillchar(indent_str[margin_tabs+1],total_spaces,' ');
    writeln(dest,concat(indent_str,copy(work_str,c_index,999)))
    end
  else
    if not skip_null_lines then writeln(dest);
  indent_level := indent_level + indent_adj
  end;

close(dest);
close(source);

{ manage file errors ** }

end.
```

End Listing

Listing One *(Text begins on page 52.)*

```
package float_unit is
type class is new float;
units_error : exception;

function "*" (left,right : class) return class;
             -- This function is to overload the inherited
             -- multiply function. Multiplying two dimensioned
             -- numbers does not produce a number with the same
             -- units, so this is an invalid operation.  It will
             -- raise the units_error exception.

-- The following multiplication functions provide for
-- multiplying a non-dimensional number (float or
-- integer) times a dimensional number (class). There
-- are two of each (one with float first, one with
-- class first) to make the multiplication functions
-- commutative.

function "*" (left : float; right : class) return class;
function "*" (left : class; right : float) return class;

function "*" (left : integer; right : class)   return class;
function "*" (left : class;   right : integer) return class;

function "/" (left,right : class) return class;
             -- This function is to overload the inherited
             -- divide function. Dividing two dimensioned numbers
             -- does not produce a number with the same units, so
             -- this is an invalid operation.  It will raise the
             -- units_error exception.

function "/" (left, right : class) return float;
             -- This function divides two items of type class and
             -- returns the result as type float.  Dividing a
             -- dimensioned number by another of the same
             -- dimensioned produces a non-dimensional number.

-- The next two divide functions allow dividing a
-- dimensioned number by a non-dimensioned floating point
-- or integer number.  Doing so produces a result with
-- the same dimensions as the dimensioned number.

function "/" (left : class; right : float) return class;
function "/" (left : class; right : integer) return class;

function "**" (left:class; right:integer) return class;
             -- This function is to overload the inherited
             -- exponentiation function. Exponentiating
             -- dimensioned numbers does not produce a
             -- number with the same units, so this is an
             -- invalid operation.  It will raise the
             -- units_error exception.

function image ( the_object :in class ) return string;
             -- This function will take the_object of type
             -- class and convert it to a string type.  The
             -- name "image" was chosen because the purpose of
             -- this function is similar to that of Ada's "image"
             -- attribute. This function and the following
             -- decouple the units package from any input/output
             -- device or package.

function value (the_string :in string) return class;
             -- This function will take a string which is a valid
             -- representation of an object of the type class and
             -- convert it to the type class.  If the_string
             -- contains an invalid value, the constraint_error
             -- exception will be raised.  The name "value" was
             -- used because the purpose of this function is
             -- similar to Ada's "value" attribute.

end float_unit;

with text_io;

package body float_unit is
-------------------------------------------------------------
function "*" (left,right : class) return class is
             -- This function is to hide the inherited multiply
             -- function. Multiplying two dimensioned numbers does
             -- not produce a number with the same units, so
             -- this is an invalid operation.  If this function
             -- is invoked, it will raise the units_error exception.

begin
             -- Whole function invalid; force exception:

     raise units_error;
     return class(float(left) * float(right));
             -- Above return needed to satisfy compiler, but
             -- it will never be executed.
end "*";
```

```
function "*" (left : float; right : class) return class
is
begin
     return class(left * float(right));
end "*";


function "*" (left : class; right : float) return class
is
begin
    return class( float(left) * right );
end "*";

function "*" (left : integer; right : class)   return
class
is
begin
    return class( float(left) * right );
end "*";

function "*" (left : class;   right : integer) return
class
is
begin
    return class( left * float(right) );
end "*";

function "/" (left,right : class) return class
is
begin
              -- Whole function invalid; force exception:

    raise units_error;
    return class( float(left) / float(right));

              -- Above return needed to satisfy compiler, but
              -- it will never be executed.
end "/";

function "/" (left, right : class) return float
is
begin
    return float(left) / float(right);
end "/";

function "/" (left : class; right : float) return class
is
begin
    return class( float(left) / right);
end "/";

function "/" (left : class; right : integer) return class
is
begin
    return class( float(left) / float(right) );
end "/";

function "**" (left:class; right:integer) return class
is
begin
    raise units_error;
    return class( float(left) ** right);
end "**";

package fio is new text_io.float_io(class);
-- Fio will be needed by image and value, below.

function image ( the_object :in class ) return string
is
    buffer : string(1..14);
begin
    fio.put(buffer, the_object);
    return buffer;
end image;
function value (the_string :in string) return class
is
    buffer : class;
    last   : positive;
begin
    fio.get(the_string, buffer, last);
    return buffer;
end value;

end float_unit;
```

End Listing One

Listing Two *(Text begins on page 52.)*

```
with float_unit;

generic
      type class_a is digits<>;
      type class_b is digits <>;
package product_unit is

type class is new float_unit.class;

function "*"(left : class_a;
             right : class_b) return class;

function "*"(left : class_b;
             right : class_a) return class;

function "/"(left : class;
             right : class_a) return class_b;

function "/"(left : class;
             right : class_b) return class_a;

end product_unit;

package body product_unit is

function "*"(left : class_a;
             right : class_b) return class
is
begin
     return class(float(left) * float(right));
end "*";

function "*"(left : class_b;
             right : class_a) return class
is
begin
     return class(float(left) * float(right));
end "*";

function "/"(left : class;
             right : class_a) return class_b
is
begin
     return class_b(float(left) / float(right));
end "/";

function "/"(left : class;
             right : class_b) return class_a
is
begin
      return class_a(float(left) / float(right));
end "/";

end product_unit;
```

Listing Three

```
with float_unit;

generic
      type numerator_class is digits <>;
      type denominator_class is digits <>;
package quotient_unit is

type class is new float_unit.class;

function "/"(left : numerator_class;
             right : denominator_class
) return class;

function "*"(left : class;
             right : denominator_class
) return numerator_class;

function "*"(left : denominator_class;
             right : class
) return numerator_class;

end quotient_unit;

package body quotient_unit is

function "/"(left : numerator_class;
             right : denominator_class) return class
is
begin
     return class(float(left) / float(right));
end "/";

function "*"(left : class;
             right : denominator_class
) return numerator_class
is
begin
     return numerator_class(float(left) * float(right));
end "*";

function "*"(left : denominator_class;
             right : class
) return numerator_class
is
begin
      return numerator_class(float(left) * float(right));
end "*";

end quotient_unit;
```

End Listings

Listing One *(Text begins on page 66.)*

```
MODULE dry;

  FROM Storage
    IMPORT ALLOCATE, DEALLOCATE, Available, InstallHeap, RemoveHeap;
  FROM Strings
    IMPORT CompareStr;

(*
 *   "DHRYSTONE" Benchmark Program
 *
 *   Version:    Mod2/1
 *   Date:       05/03/86
 *   Author:       Reinhold P. Weicker,  CACM Vol 27, No 10, 10/84 pg. 1013
 *         C version translated from ADA by Rick Richardson
 *         Every method to preserve ADA-likeness has been used,
 *         at the expense of C-ness.
 *         Modula-2 version translated from C by Kevin Northover.
 *         Again every attempt made to avoid distortions of the original.
 *   Machine Specifics:
 *         The time function is system dependant, one is
 *         provided for the Amiga.  Your compiler may be different.
 *         The LOOPS constant is initially set for 50000 loops.
 *         If you have a machine with large integers and is
 *         very fast, please change this number to 500000 to
 *         get better accuracy.
 *         You can also time the program with a stopwatch when it
 *         is lightly loaded (no interlaced 4 bit deep Amiga screens ...).
 *
 ***************************************************************************
 *
 *   The following program contains statements of a high-level programming
 *   language (Modula-2) in a distribution considered representative:
 *
 *   assignments         53%
 *   control statements      32%
 *   procedure, function calls   15%
 *
 *   100 statements are dynamically executed.  The program is balanced with
 *   respect to the three aspects:
 *      - statement type
 *      - operand type (for simple data types)
 *      - operand access
 *           operand global, local, parameter, or constant.
 *
 *   The combination of these three aspects is balanced only approximately.
 *
 *   The program does not compute anything meaningfull, but it is
 *   syntactically and semantically correct.
 *
 *)

(* Accuracy of timings and human fatigue controlled by next two lines *)

  CONST
    LOOPS = 50000;

  TYPE
    Enumeration = (Ident1, Ident2, Ident3, Ident4, Ident5);
    OneToThirty = CARDINAL;
    OneToFifty = CARDINAL;
    CapitalLetter = CHAR;
    String30 = ARRAY [0..30-1] OF CHAR;
    Array1Dim = ARRAY [0..50] OF CARDINAL;
    Array2Dim = ARRAY [0..50], [0..50] OF CARDINAL;
    RecordPtr = POINTER TO RecordType;
    RecordType = RECORD
                   PtrComp: RecordPtr;
                   Discr: Enumeration;
                   EnumComp: Enumeration;
                   IntComp: OneToFifty;
                   StringComp: String30;
                 END;

    (*
     * Package 1
     *)

  VAR

    IntGlob: CARDINAL;
    BoolGlob: BOOLEAN;
    Char1Glob: CHAR;
    Char2Glob: CHAR;
    Array1Glob: Array1Dim;
    Array2Glob: Array2Dim;
    PtrGlb: RecordPtr;
    PtrGlbNext: RecordPtr;


  PROCEDURE Proc7(IntParI1, IntParI2: OneToFifty;
                  VAR IntParOut: OneToFifty);
    VAR

      IntLoc: OneToFifty;
BEGIN
  IntLoc := IntParI1+2;
  IntParOut := IntParI2+IntLoc;
END Proc7;


PROCEDURE Proc3(VAR PtrParOut: RecordPtr);
BEGIN
  IF (PtrGlb <> NIL) THEN

    PtrParOut := PtrGlb^.PtrComp
  ELSE
    IntGlob := 100
  END;
  Proc7(10, IntGlob, PtrGlb^.IntComp);
END Proc3;


PROCEDURE Func3(EnumParIn: Enumeration): BOOLEAN;
    VAR
      EnumLoc: Enumeration;
    VAR Func3Result: BOOLEAN;
  BEGIN
    EnumLoc := EnumParIn;
    Func3Result := EnumLoc = Ident3;
    RETURN Func3Result
  END Func3;


  PROCEDURE Proc6(EnumParIn: Enumeration;
                  VAR EnumParOut: Enumeration);
  BEGIN
    EnumParOut := EnumParIn;
    IF ( NOT Func3(EnumParIn)) THEN
      EnumParOut := Ident4
    END;
    CASE EnumParIn OF
        Ident1:
        EnumParOut := Ident1
      | Ident2:
        IF (IntGlob > 100) THEN

          EnumParOut := Ident1
        ELSE
          EnumParOut := Ident4
        END
      | Ident3:
        EnumParOut := Ident2
      | Ident4:
      | Ident5:
        EnumParOut := Ident3

      ELSE
    END;
  END Proc6;



  PROCEDURE Proc1(PtrParIn: RecordPtr);
  BEGIN
    WITH PtrParIn^ DO

      PtrComp^ := PtrGlb^;
      IntComp := 5;
      PtrComp^.IntComp := IntComp;
      PtrComp^.PtrComp := PtrComp;
      Proc3(PtrComp^.PtrComp);
      IF (PtrComp^.Discr = Ident1) THEN
        PtrComp^.IntComp := 6;
        Proc6(EnumComp, PtrComp^.EnumComp);
        PtrComp^.PtrComp := PtrGlb^.PtrComp;
        Proc7(PtrComp^.IntComp, 10, PtrComp^.IntComp);


      ELSE
        PtrParIn^ := PtrComp^
      END;
    END;
  END Proc1;


  PROCEDURE Proc2(VAR IntParIO: OneToFifty);

    VAR

      IntLoc: OneToFifty;
      EnumLoc: Enumeration;
  BEGIN
    IntLoc := IntParIO+10;
    REPEAT
    IF (Char1Glob = 'A') THEN

      DEC(IntLoc, 1);
      IntParIO := IntLoc-IntGlob;
      EnumLoc := Ident1;
    END;
  UNTIL EnumLoc = Ident1;
END Proc2;


PROCEDURE Proc4;

  VAR

    BoolLoc: BOOLEAN;
BEGIN
  BoolLoc := Char1Glob = 'A';
  BoolLoc := BoolLoc OR BoolGlob;
  Char2Glob := 'B';
END Proc4;


PROCEDURE Proc5;
BEGIN
  Char1Glob := 'A';
  BoolGlob := FALSE;
```

```
END Proc5;


PROCEDURE Proc8(VAR Array1Par: Array1Dim;
                VAR Array2Par: Array2Dim;
                IntParI1, IntParI2: OneToFifty);

  VAR

    IntLoc: OneToFifty;
    IntIndex: OneToFifty;
BEGIN
  IntLoc := IntParI1+5;
  Array1Par[IntLoc] := IntParI2;
  Array1Par[IntLoc+1] := Array1Par[IntLoc];
  Array1Par[IntLoc+30] := IntLoc;
  FOR IntIndex := IntLoc TO (IntLoc+1) DO
    Array2Par[IntLoc][IntIndex] := IntLoc
  END;
  Array2Par[IntLoc][IntLoc-1] := Array2Par[IntLoc][IntLoc-1]+1;
  Array2Par[IntLoc+20][IntLoc] := Array1Par[IntLoc];
  IntGlob := 5;
END Proc8;


PROCEDURE Func1(CharPar1, CharPar2: CapitalLetter): Enumeration;

  VAR

    CharLoc1, CharLoc2: CapitalLetter;
  VAR Func1Result: Enumeration;
BEGIN
  CharLoc1 := CharPar1;
  CharLoc2 := CharLoc1;
  IF (CharLoc2 <> CharPar2) THEN
    Func1Result := (Ident1)
  ELSE
    Func1Result := (Ident2)
  END;
  RETURN Func1Result
END Func1;


PROCEDURE Func2(VAR StrParI1, StrParI2: String30): BOOLEAN;

  VAR

    IntLoc: OneToThirty;
    CharLoc: CapitalLetter;
  VAR Func2Result: BOOLEAN;
BEGIN
  IntLoc := 2;
  WHILE (IntLoc <= 2) DO
    IF (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) = Ident1) THEN
      CharLoc := 'A';
      INC(IntLoc, 1);
    END;
  END;
  IF (CharLoc >= 'W') AND (CharLoc <= 'Z') THEN
    IntLoc := 7
  END;
  IF CharLoc = 'X' THEN
    Func2Result := TRUE
  ELSIF CompareStr (StrParI1, StrParI2) > 0 THEN
    INC(IntLoc, 7);
    Func2Result := TRUE
  ELSE
    Func2Result := FALSE
  END;
  RETURN Func2Result
END Func2;



PROCEDURE Proc0;

  VAR

    IntLoc1: OneToFifty;
    IntLoc2: OneToFifty;
    IntLoc3: OneToFifty;
    CharLoc: CHAR;
    CharIndex: CHAR;
    EnumLoc: Enumeration;
    String1Loc, String2Loc: String30;
    i, LoopMax: CARDINAL;


BEGIN
  LoopMax := LOOPS;
  NEW(PtrGlbNext);
  NEW(PtrGlb);
  PtrGlb^.PtrComp := PtrGlbNext;
  PtrGlb^.Discr := Ident1;
  PtrGlb^.EnumComp := Ident3;
  PtrGlb^.IntComp := 40;
  PtrGlb^.StringComp := 'DHRYSTONE PROGRAM, SOME STRING';
  String1Loc := "DHRYSTONE PROGRAM, 1'ST STRING";
  FOR i := 0 TO LoopMax DO

    Proc5;
    Proc4;
    IntLoc1 := 2;
    IntLoc2 := 3;
    String2Loc := "DHRYSTONE PROGRAM, 2'ND STRING";
    EnumLoc := Ident2;
    BoolGlob :=  NOT Func2(String1Loc, String2Loc);
    WHILE (IntLoc1 < IntLoc2) DO

      IntLoc3 := 5*IntLoc1-IntLoc2;
      Proc7(IntLoc1, IntLoc2, IntLoc3);
      INC(IntLoc1, 1);
    END;
    Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
    Proc1(PtrGlb);
    CharIndex := 'A';
    WHILE CharIndex <= Char2Glob DO

      IF (EnumLoc = Func1(CharIndex, 'C')) THEN
        Proc6(Ident1, EnumLoc)
      END;
      CharIndex := VAL(CHAR, ORD(CharIndex)+1);
    END;
    IntLoc3 := IntLoc2*IntLoc1;
    IntLoc2 := IntLoc3 DIV IntLoc1;
    IntLoc2 := 7*(IntLoc3-IntLoc2)-IntLoc1;
      Proc2(IntLoc1);
    END;
 END Proc0;



  (* The Main Program is trivial *)

BEGIN
  Proc0;
END dry.
```

End Listing One

Listing Two *(Listing continued, text begins on page 66.)*

```
MODULE sieve;
(* Eratosthenes sieve prime number program, Byte Magazine *)

  CONST size = 8190;

  VAR
    psn, k, prime, iter : INTEGER;
    flags : ARRAY [0..size] OF BOOLEAN;

BEGIN
  FOR iter := 1 TO 25 DO
    FOR psn := 0 TO size DO
              flags[ psn ] := TRUE;
    END(* for *);
    FOR psn := 0 TO size DO
              IF flags[ psn ]
              THEN  (* prime *)
                prime := psn + psn + 3;
                k := psn + prime;
                WHILE k <= size DO  (* cancel multiples *)
                            flags[ k ] := FALSE;
                            k := k + prime;
                END(* while *);
              END(* if then *);
    END(* for *);
  END(* for *);
END sieve.
```

End Listing Two

Listing Three

```
MODULE fib;

(* Berkeley standard benchmark *)
(* Computes largest 16-bit Fibonacci number *)
(* Tests compiler recursion efficiency and CPU thruput *)

  CONST
    TIMES = 10;
    VALUE = 24;

  VAR
    i: INTEGER;
    f: CARDINAL;
    (* ------------------------------------------------------- *)

  PROCEDURE fibonacci(n: INTEGER): CARDINAL;
    VAR fibonacciResult: CARDINAL;
  BEGIN
    IF n >= 2 THEN
      fibonacciResult := fibonacci(n-1)+fibonacci(n-2)
    ELSE
      fibonacciResult := n
    END;
    RETURN fibonacciResult
  END fibonacci; (* -------------------------- *)


BEGIN (* main *)
  FOR i := 1 TO TIMES DO
    f := fibonacci(VALUE)
  END;
END fib.
```

End Listing Three

Listing Four

```
MODULE acker;

(* Berkeley standard benchmark *)
(* Ackerman's function: ack (2, 4) *)
(* Tests recursion and integer math *)
(* Repeats 10,000 times *)



  VAR
    loop, r: INTEGER;
    (* ------------------------------------------------------------ *)



  PROCEDURE ack(x1, x2: INTEGER): INTEGER;

    VAR
      result: INTEGER;

    VAR ackResult: INTEGER;
  BEGIN
    IF x1 = 0 THEN

      result := x2+1
    ELSIF x2 = 0 THEN
      result := ack(x1-1, 1)
    ELSE
      result := ack(x1-1, ack(x1, x2-1))
    END;
    ackResult := result;
    RETURN ackResult
  END ack; (* -------------------------- *)


BEGIN (* main *)
  FOR loop := 1 TO 10000 DO
    r := ack(2, 4)
  END;
END acker.
```

End Listing Four

Listing Five

```
MODULE FPMath;
(* Benchmarks floating point math package *)

  FROM MathLib0 IMPORT arctan, exp, ln, sin, sqrt;
  FROM InOut    IMPORT Write, WriteLn, WriteString;

  CONST
    pi = 3.1415927;
    nloops = 5;

  VAR
    i, j: INTEGER;
    angle, result, argument: REAL;

BEGIN
  WriteString('SQUARE ROOTS   ');
  FOR i := 1 TO nloops DO
    Write ('.');
    argument := 0.0;
    WHILE argument <= 1000.0 DO
      result := sqrt (argument);
      argument := argument + 1.0
    END;
  END; (* FOR *)

  WriteLn;
  WriteString('LOGS           ');
  FOR i := 1 TO nloops DO
    Write ('.');
    argument := 0.1;
    WHILE argument <= 1000.1 DO
      result := ln (argument);
      argument := argument + 1.0
    END;
  END; (* FOR *)

  WriteLn;
  WriteString('EXPONENTIALS   ');
  FOR i := 1 TO nloops DO
    Write ('.');
    argument := 0.1;
    WHILE argument <= 10.0 DO
      result := exp (argument);
      argument := argument + 0.01
    END;
  END; (* FOR *)

  WriteLn;
  WriteString('ARCTANS        ');
  FOR i := 1 TO nloops DO
    Write ('.');
    argument := 0.1;
    WHILE argument <= 10.0 DO
      angle := arctan (argument);
      argument := argument + 0.01
    END;
  END; (* FOR *)

  WriteLn;
  WriteString('SINES          ');
  FOR i := 1 TO nloops DO
    Write ('.');
    angle := 0.0;
    WHILE angle <= 2.0 * pi DO
      result := sin (angle);
      angle := angle + pi / 360.0
    END;
  END; (* FOR *)
  WriteLn;
END FPMath.
```

End Listing Five

Listing Six *(Text begins on page 66.)*

```
MODULE QSort;

(* The test uses QuickSort to measure recursion speed *)
(* An ordered array is created by the program and is  *)
(* reverse sorted.  The process is performed 'MAXITER'*)
(* number of times.                                   *)

CONST SIZE = 1000;
      MAXITER = 50;

TYPE NUMBERS = ARRAY[1..SIZE] OF CARDINAL;

VAR Iter, Offset, I, J, Temporary : CARDINAL;
    A : NUMBERS;

PROCEDURE InitializeArray ;
(* Procedure to initialize array *)

VAR I : CARDINAL;

BEGIN
    FOR I := 1 TO SIZE DO
        A[I] := SIZE - I + 1
    END; (* FOR I *)
END InitializeArray;

PROCEDURE QuickSort;
(* Procedure to perform a QuickSort *)

PROCEDURE Sort(Left, Right : CARDINAL);

VAR i, j : CARDINAL;
    Data1, Data2 : CARDINAL;

BEGIN
    i := Left; j := Right;
    Data1 := A[(Left + Right) DIV 2];
    REPEAT
        WHILE A[i] < Data1 DO INC(i) END;
        WHILE Data1 < A[j] DO DEC(j) END;
        IF i <= j THEN
            Data2 := A[i]; A[i] := A[j]; A[j] := Data2;
            INC(i); DEC(j)
        END;
    UNTIL i > j;
    IF Left < j  THEN Sort(Left,j)  END;
    IF i < Right THEN Sort(i,Right) END;
END Sort;

BEGIN (* QuickSort *)
    Sort(1,SIZE);
END QuickSort;

BEGIN (* Main *)
    FOR Iter := 1 TO MAXITER  DO
       InitializeArray;
       QuickSort
    END; (* FOR Iter  *)
END QSort.
```

End Listing Six

Listing Seven

```
MODULE ShSort;
(* Tests Shell sort speed on an integer array of ARSIZE elements.  *)
(* Creates an array ordered from smaller to larger, then sorts it  *)
(* into reverse order. Repeats NSORTS times.                       *)

CONST ARSIZE = 1000;
      NSORTS = 20;

TYPE NUMBERS = ARRAY [1..ARSIZE] OF INTEGER;

VAR IsInOrder, Ascending : BOOLEAN;
    Iter, Offset, I, J, Temporary : CARDINAL;
    Ch : CHAR;
    A : NUMBERS;

PROCEDURE InitializeArray ;
     (* Initialize array *)
BEGIN
    FOR I := 1 TO ARSIZE DO
        A [I] := I
    END; (* FOR I *)
```

```
END InitializeArray;

PROCEDURE ShellSort ;
     (* Shell-Meztner sort *)

    PROCEDURE Swap;
         (* Swap elements A[I] and A[J] *)
    BEGIN
       IsInOrder := FALSE;
       Temporary := A[I];
       A[I] := A[J];
       A[J] := Temporary;
    END Swap;

BEGIN
   (* Toggle 'Ascending' flag *)
      Ascending := NOT Ascending;
      Offset := ARSIZE;
      WHILE Offset > 1 DO
          Offset := Offset DIV 2;
          REPEAT
              IsInOrder := TRUE;
              FOR J := 1 TO (ARSIZE - Offset) DO
                  I := J + Offset;
                  IF Ascending
                      THEN IF A[I] < A[J] THEN Swap END
                      ELSE IF A[I] > A[J] THEN Swap END
                  END; (* IF AscendingOrder *)
              END; (* FOR J *)
          UNTIL IsInOrder;
       END; (* End of while-loop *)
END ShellSort;

BEGIN (* Main *)
    InitializeArray;
    Ascending := TRUE;
    FOR Iter := 1 TO NSORTS DO
       ShellSort
    END;
END ShSort.
```

End Listing Seven

Listing Eight

```
MODULE cortn;

(* Benchmark to test speed of coroutine switching *)
(* Shifts NCHARS characters to upper-case         *)
(* Two transfers per character                    *)

FROM SYSTEM IMPORT NEWPROCESS, TRANSFER, ADDRESS, BYTE, ADR;

CONST  NCHARS = 50000;
       WorkSize = 1000;
VAR    ch : ARRAY [1..NCHARS] OF CHAR;
       ShiftWork, CountWork : ARRAY [1..WorkSize] OF BYTE;
       count, chval, c : CARDINAL;
       main, shifter, counter : ADDRESS;

PROCEDURE CountProc;
    (* Increments count *)
BEGIN
  REPEAT
    count := count + 1;
    TRANSFER (counter, shifter);
  UNTIL FALSE;
END CountProc;

PROCEDURE ShiftProc;
    (* Shifts char at 'count' to upper case *)
BEGIN
  REPEAT
    IF (ch [count] >= 'a') AND (ch [count] <= 'z') THEN
      ch [count] := CHR (ORD (ch [count]) - 32)
    END;
    TRANSFER (shifter, counter);
  UNTIL count = NCHARS;
  TRANSFER (shifter, main);
END ShiftProc;

BEGIN  (* Main program *)

  (* Load array with lower-case letters *)
  chval := ORD ('a');
  FOR c := 1 TO NCHARS DO
    ch [c] := CHR (chval);
    chval := chval + 1;
    IF chval > ORD ('z') THEN
      chval := ORD ('a');
    END;
  END;

  (* Set up coroutines *)
  NEWPROCESS (CountProc, ADR (CountWork), WorkSize, counter);
  NEWPROCESS (ShiftProc, ADR (ShiftWork), WorkSize, shifter);

  (* Dispatch the controlling task *)
  count := 1;
  TRANSFER (main, shifter);
END cortn.
```

End Listing Eigh

Listing Nine

```
MODULE ncortn;

(* Does the same thing as CORTN.MOD, but without  *)
(* coroutine switching                            *)
(* Subtract run time for this from time for CORTN *)
(* to find out actual coroutine overhead          *)

CONST  NCHARS = 50000;
       WorkSize = 1000;

VAR    ch : ARRAY [1..NCHARS] OF CHAR;
       count, chval, c : CARDINAL;

PROCEDURE CountProc;
    (* Increments count *)
BEGIN
  count := count + 1;
END CountProc;

PROCEDURE ShiftProc;
    (* Shifts all chars in array 'ch' upper case *)
BEGIN
  REPEAT
    IF (ch [count] >= 'a') AND (ch [count] <= 'z') THEN
      ch [count] := CHR (ORD (ch [count]) - 32)
    END;
    CountProc;          (* Substitute call for TRANSFER *)
  UNTIL count = NCHARS;
END ShiftProc;

BEGIN  (* Main program *)

  (* Load array with lower-case letters *)
  chval := ORD ('a');
  FOR c := 1 TO NCHARS DO
    ch [c] := CHR (chval);
    chval := chval + 1;
    IF chval > ORD ('z') THEN
      chval := ORD ('a');
    END;
  END;

  (* Dispatch the controlling task *)
  count := 1;
  ShiftProc;
END ncortn.
```

End Listings

HYPERCARD'S SOFTWARE SLOTS

Listing One *(Text begins on page 90.)*

```
*------------------------------------- file information
*     XCMDandXFCN.a
*     MPW record definitions useful in XCMD and XFCN work
*     Edited with MPW 2.02
*     Compiled under MPW 2.02
*     Written and (C)1988 by Stan Krute. All rights reserved.
*-------------------------------------------------------------------------

; general array structures

Array4L         RECORD    0           ; an array of 4 longs
zero            DS.L      1
one             DS.L      1
two             DS.L      1
three           DS.L      1
                ENDR

Array8L         RECORD    0           ; an array of 8 longs
zero            DS.L      1
one             DS.L      1
two             DS.L      1
three           DS.L      1
four            DS.L      1
five            DS.L      1
six             DS.L      1
seven           DS.L      1
                ENDR

Array16L        RECORD    0           ; an array of 16 longs
zero            DS.L      1
one             DS.L      1
two             DS.L      1
three           DS.L      1
four            DS.L      1
five            DS.L      1
six             DS.L      1
seven           DS.L      1
eight           DS.L      1
nine            DS.L      1
ten             DS.L      1
eleven          DS.L      1
twelve          DS.L      1
thirteen        DS.L      1
fourteen        DS.L      1
fifteen         DS.L      1
                ENDR

; an XCmdBlock record

XCmdBlock       RECORD    0
paramCount      DS.W      1           ; # of entry parameters
params          DS        Array16L    ; array of handles to entry parameters
returnValue     DS.L      1           ; handle to a return value
passFlag        DS.W      2           ; boolean: if true, HC passes message on
```

```
entryPoint      DS.L        1           ; pointer to HyperCard callback function (cbf)
request         DS.W        1           ; a cbf command code
result          DS.W        1           ; HC's answer to a cbf command
inArgs          DS          Array8L     ; array of arguments in to a cbf command
outArgs         DS          Array4L     ; array of arguments out from a cbf command
                ENDR
```

Listing Two

End Listing One

```
# make instructions for the isAlpha XFCN

# the object file depends on the assembly language source code file
isAlpha.a.o     f     isAlpha.a
# to create the object file, assemble the assembly language source code file
     # the source code file is isAlpha.a
     # we'll send a listing to the file isAlpha.a.lst
     Asm isAlpha.a -l

# the resource file depends on the object file
isAlpha     f     isAlpha.a.o
# to create the resource file, link the object file
     # the output file is isAlpha
     # the output file's type is RSRC
     # the output file's creator is RSED
     # the code will be an XFCN resource with ID #990
     # the segment name will be isAlpha, same as the new HyperCard command name
     # the input file is the object file isAlpha.a.o
     Link -o isAlpha -t RSRC -c RSED -rt XFCN=990 -sn Main=isAlpha isAlpha.a.o
```

Listing Three

End Listing Two

```
*------------------------------------- file information
*     isAlpha.a
*     Assembly language source code for the HyperCard XFCN isAlpha
*     Given a character, isAlpha returns the string 'true'
*       if the character is an uppercase or lowercase letter, the
*       string 'false' if it is not.
*       A c function prototype might look like this:
*          BOOLEAN_STRING isAlpha(CHAR chSomeChar) ;
*     Edited with MPW 2.02
*     Compiled under MPW 2.02
*     Written and )1988 by Stan Krute. All rights reserved.
*       No part of this file other files in the project it belongs to,
*       or the object code the file(s) lead(s) to, may be reproduced,
*       in any form or by any means, without the express written
*       permission of the author and copyright holder.
*-----------------------------------------------------------------

*------------------------------------- include files

; standard Mac definitions
     INCLUDE     'QuickEqu.a'       ; quickdraw
     INCLUDE     'ToolEqu.a'        ; toolbox
     INCLUDE     'SysEqu.a'         ; system
     INCLUDE     'Traps.a'          ; traps

; our stuff
     INCLUDE     'XCMDandXFCN.a'    ; XCMD and XFCN definitions

*-------------------------------------- isAlpha -----------------

; set local constants
smallBlockSize              SET     6     ; size of a small result block we'll allocate
XCmdPtr                     SET     8     ; where we'll find the XCmdPtr after register saving
bytesOfEntryParams          SET     4     ; bytes of entry parameters

isAlpha     MAIN
            ; save a register
            MOVE.L          A2,-(SP)

            ; get a small result block, filled with zeroes for cheap 0-terminated strings
            MOVE.L          #smallBlockSize,D0
            _NewHandle      ,CLEAR

            ; store the result block's handle as a return value
            MOVE.L          XCmdPtr(SP),A2              ; A2 gets pointer to XCmdBlock
            MOVE.L          A0,XCmdBlock.returnValue(A2)    ; handle to result block

            ; point to the result block
            MOVE.L          (A0),A0

            ; get the single entry parameter
            MOVE.L          XCmdBlock.params.zero(A2),A1    ; handle to first entry parameter
            MOVE.L          (A1),A1                     ; pointer to first entry parameter

            ; clear a data register, and move entry parameter, a character, into it
            CLR.L           D0
            MOVE.B          (A1),D0

            ; now run a series of boundary tests
            ; check against the low bound of upper-casedness
            CMP.B           #'A',D0
            BLT.S           notAlpha        ; less than this, and we're not alphabetical

            ; check against the high bound of upper-casedness
            CMP.B           #'Z',D0
            BLE.S           yesAlpha        ; less than or equal to this, and we're upper-case,
                                            ; thus alphabetical

            ; check against the low bound of lower-casedness
            CMP.B           #'a',D0
            BLT.S           notAlpha        ; less than this, and we're not alphabetical

            ; check against the high bound of lower-casedness
            CMP.B           #'z',D0
            BGT.S           notAlpha        ; greater than this, and we're not
                                              alphabetical
yesAlpha
            ; we have an upper- or lower-case letter
            ; set result into result block
            MOVE.L          #'true',(A0)
            BRA.S           bye
notAlpha
            ; we have neither an upper- nor a lower-case letter
            ; set result into result block
            MOVE.L          #'fals',(A0)+
            MOVE.B          #'e',(A0)

bye         ; tell HyperCard we've handled things
            CLR.W           XCmdBlock.passFlag(A2)

            ; restore a register
            MOVE.L          (SP)+,A2

            ; move the return address into A0
            MOVE.L          (A7)+,A0

            ; jump over entry parameters
            ADD.W           #bytesOfEntryParams,A7

            ; puffasmoke and we be gone
            JMP             (A0)

            ; end of the procedure
            ENDPROC

            ; end of the assembly language source file
```

End Listings

ASSEMBLY LANGUAGE

Listing One *(Text begins on page 82.)*

```
1    page 60,132
2    title FILE_LENGTH - Determine File Length In Bytes
3    name FILE_LEN
4
5  comment @
6    FILE_LENGTH()                                          V 1.00
7  ------------------------------------------------------------
8  NAME
9    file_length()          Determine file length in bytes
10
11   SYNOPSIS
12     length = file_length(filename, attribute);
13     long          length;          length of file in bytes
14     char          *filename;       path\filename.ext
15     unsigned int  attribute;       file attribute used
16                                     in search
17
18   DESCRIPTION
19     This function uses DOS function 4eh, Find First
20     File, to return with the file length.  So that this
21     function does not corrupt the current Data Transfer
22     Area, DTA, this function moves the DTA to its own
23     automatic variable area.  If the file is not found,
24     -1L is returned.
25
26     This function allows any kind of file to be found by
27     specifying the file attribute.  The following #define
28     statements can be used to define the various file
29     attributes:
30
31     #define NORM          0x00    /* Normal file    */
32     #define RO            0x01    /* Read-only file */
33     #define HID           0x02    /* Hidden file    */
34     #define SYS           0x04    /* System file    */
35     #define VOL_ID        0x08    /* Volume ID      */
36     #define SUBDIR        0x10    /* Subdirectory   */
37     #define ARCH          0x20    /* Archive file   */
38
39     The bit or operator, |, can be used to combine
40     several attributes.  For example:
41
42     length = file_length("filename.ext", RO|HID|SYS);
43
44     All normal, read-only, hidden, and system files will
45     be searched for a match to "filename.ext."
46
47     This procedure was assembled using MICROSOFT MASM
48     V5.0 with the switches, /v /ml /z, set.
49
50   CAUTION
51     None.
52
53   RETURNS
54     File length as a long.  -1L signifies an error.
55
56   AUTHOR
57     Raymond Moon - 18 Nov 87
58
59     Copyright (c) 1987, MoonWare
60     ALL RIGHTS RESERVED
61
62   HISTORY
63     Version - Date         Remarks
64     1.00  - 18 Nov 87      - Orginal
65
66     @
67
68   ;----------------------------
69   ;  Define the segments for the memory model so that
70   ;  they can link with Microsoft C.
71
72   _TEXT   segment byte public 'CODE'
73   _TEXT   ends
74
75   _DATA   segment word    public  'DATA'
76   _DATA   ends
77
78   CONST   segment word    public  'CONST'
79   CONST   ends
80
81   _BSS    segment word    public  'BSS'
```

```
_BSS    ends

DGROUP  group   _DATA, CONST, _BSS

;----------------------------
;  Define the assume directive so MASM knows to what
;  the segment registers point

assume  cs:_TEXT, ds:DGROUP, ss:DGROUP, es:DGROUP

;----------------------------
;  Define the structures for passed parameters and
;  automatic variables.

PARMS    struc
         dw     2 dup (?)
FILENAME dw     ?
ATTR     dw     ?
PARMS    ends

;----------------------------
;  _AUTO structure contains the following variables:
;  OLD_DTA      Saved address of old DTA
;  NEW_DTA      Start of NEW_DTA, this first field is
;                reserved for subseqent Find Next File
;  F_ATTR       Attribute found
;  F_TIME       Time file was last written
;  F_DATE       Date file was last written
;  F_LENGTH     Double word file length
;  F_NAME       Found filename.ext

_AUTO    struc
OLD_DTA  dd     ?
NEW_DTA  db     21 dup (?)
F_ATTR   db     ?
F_TIME   dw     ?
F_DATE   dw     ?
F_LENGTH dd     ?
F_NAME   db     13 dup (?)
_AUTO    ends

;----------------------------
;  Define structures for addressing the offset and
;  segment portions of doubleword pointers and the
;  least and most significant words of long integers.

DOUBLEWORD_PTR  struc
OFF     dw      ?
SEGMT   dw      ?
DOUBLEWORD_PTR  ends

LONG_INT        struc
LSW     dw      ?
MSW     dw      ?
LONG_INT        ends

;----------------------------
;  Start the Code Segment.

_TEXT   segment

_FILE_LENGTH    proc    near

        public  _FILE_LENGTH

;----------------------------
;  Standard entry logic.  Save calling BP.  Set up new
;  BP.  Define automatic variable addressing base,
;  AUTO.  Save segment registers as DS and
;  ES are used.

        push    bp
        mov     bp,sp
        sub     sp, size _AUTO
        AUTO    equ [bp - size _AUTO]
        push    ds
        push    es

;----------------------------
;  Get and save current DTA address in OLD_DTA.  Use
;  DOS service 2fh.  The address is returned in ES:BX.

        mov     ah,2fh
        int     21h
        mov     AUTO.OLD_DTA.OFF,bx
        mov     AUTO.OLD_DTA.SEGMT,es

;----------------------------
;  Set up NEW_DTA as the current DTA.  Use DOS service
;  1ah.  DS = SS assumed.

        mov     ah,1ah
        lea     dx,AUTO.NEW_DTA
        int     21h

;----------------------------
;  Set up for File First Find, Service 4eh.
;  CX = Search attribute
;  DS:DX => ASCIIZ string, d:path\filename.ext

        mov     ah,4eh
        mov     cx,[bp].ATTR
        mov     dx,[bp].FILENAME
        int     21h

;----------------------------
;  Check for Carry Flag set as that indicates that the
;  file was not found.  If so, put -1L in F_LENGTH so
;  -1L is returned as file length.

        jnc     FL1
        mov     ax,-1
        mov     AUTO.F_LENGTH.LSW,ax
        mov     AUTO.F_LENGTH.MSW,ax

;----------------------------
;  Restore old DTA, service 1ah.
;  DS:DX => old DTA.

FL1:    mov     ah,1ah
        mov     dx,AUTO.OLD_DTA.OFF
        mov     ds,AUTO.OLD_DTA.SEGMT
        int     21h

;----------------------------
;  Set up AX:DX to return file length.

        mov     ax,AUTO.F_LENGTH.LSW
        mov     dx,AUTO.F_LENGTH.MSW

;----------------------------
;  Exit logic.  Restore segment registers.  Remove any
;  automatic variables.  Restore calling BP.

        pop     es
        pop     ds
        mov     sp,bp
        pop     bp
        ret

_FILE_LENGTH    endp

_TEXT   ends

        end
```

End Listing

C PROGRAMMING

Listing One *(Text begins on page 116.)*

```
/* ---------- window.h ----------- */

void establish_window(int,int,int,int,int,int,int);
void window_title(char *);
void clear_window(void);
void delete_window(void);
void scroll_window(int);
void text_window(char **, int);
int select_window(int, int, int, int (*func)(int,int));
int getkey(void);
void hidecursor(void);
void set_cursor_type(unsigned);
void clear_screen(void);
void writeline(int, int, char *);
void current_window(void);
void error_message(char *);

#define MAX_WINDOWS 10      /* maximum windows open at once */

#define TRUE         1
#define FALSE        0
#define ERROR       -1

#define BELL         7
#define ESC         27
#define SHIFT_HT   143
#define CTRL_T      20
#define CTRL_B       2
#define CTRL_D       4
#define ALT_D      160
#define ALT_F      161
#define ALT_E      146
#define ALT_O      152
#define ALT_S      159

#define F1         187
#define F2         188
#define F3         189
#define F4         190
#define F5         191
#define F6         192
#define F7         193
#define F8         194
#define F9         195
#define F10        196
#define ALT_F7     238
```

```
#define HOME       199
#define UP         200
#define PGUP       201
#define BS         203
#define FWD        205
#define END        207
#define DN         208
#define PGDN       209
#define INS        210
#define DEL        211

#define CTRL_HOME 247
#define CTRL_BS   243
#define CTRL_FWD  244
#define CTRL_END  245

/* --------- window definition structure ------------ */
struct wn {
            int lf,tp,rt,bt;        /* window position */
            int ht,wd;              /* window dimensions */
            int wx, wy;             /* window cursor */
            int wtop;               /* top text line */
            int wlines;             /* total text lines */
            int fg,bg;              /* window colors */
            char *wsave;            /* video memory save buffer */
            char **wtext;           /* pointer to text */
};

/* ------ internal Turbo C stuff ------- */
void far * pascal __vptr(int, int);
void pascal __vram(void far *, void far *, int);
extern struct {
            char filler1[4];
            char attribute;
            char filler2[5];
            char snow;
} _video;

/* ------------ window colors --------------- */
#define TEXTFG  WHITE       /* data display screen */
#define TEXTBG  BLACK
#define BLOCKFG BLACK       /* data blocks */
#define BLOCKBG WHITE
#define HELPBG  WHITE       /* help windows */
#define HELPFG  BLACK
#define MENUBG  WHITE       /* menus */
#define MENUFG  BLACK
#define SELECTBG BLACK      /* menu selector bars */
#define SELECTFG WHITE
#define ENTRYFG WHITE       /* data entry windows */
#define ENTRYBG BLACK
#define FIELDFG BLACK       /* data entry fields  */
#define FIELDBG WHITE
#define ERRORFG BLACK       /* error messages */
#define ERRORBG WHITE
```

End Listing One

Listing Two

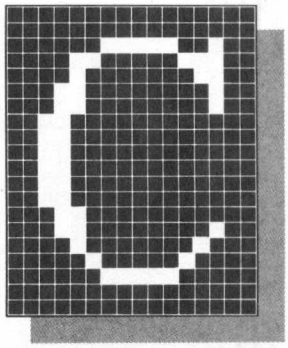
```
/* ----------------------- window.c --------------------- */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <conio.h>
#include <mem.h>
#include <dos.h>
#include <stdlib.h>
#include "window.h"

/* --------- window border characters ---------------- */
#define NW   '\332'
#define NE   '\277'
#define SE   '\331'
#define SW   '\300'
#define SIDE '\263'
#define LINE '\304'

int editing;
static union REGS rg;

/* --------- window definition structure ------------ */
struct wn wdo [MAX_WINDOWS];
int curr_wnd;   /* current window */
struct wn wkw;   /* a working window structure */

static void upline(void);
static void downline(void);
static void firstline(void);
static void lastline(void);
static void dline(int, int, int);

/* ----------- establish a new window -------------- */
void establish_window(left,top,right,bottom,foreg,backg,save)
{
   if (curr_wnd < MAX_WINDOWS)   {
      if (curr_wnd)
         wdo[curr_wnd-1] = wkw;
      setmem(&wkw, sizeof(wkw), 0);
      wkw.lf = left;
      wkw.tp = top;
      wkw.rt = right;
      wkw.bt = bottom;
      wkw.fg = foreg;
      wkw.bg = backg;
      wkw.wd = right+1-left;
      wkw.ht = bottom-top-1;
      if (save)   {
         if ((wkw.wsave=malloc((wkw.ht+2)*wkw.wd*2)) == NULL)
            return;
         gettext(left, top, right, bottom, wkw.wsave);
      }
      wdo[curr_wnd++] = wkw;
      current_window();
      clear_window();
   }
}

/* ------- initialize the working window as current -------- */
void current_window()
{
   window(wkw.lf,wkw.tp,wkw.rt,wkw.bt);
   hidecursor();
   if (wkw.fg || wkw.bg)   {
      textcolor(wkw.fg);
      textbackground(wkw.bg);
    }
}

/* ----------- set a window's title -------------- */
void window_title(char *ttl)
{
   writeline((wkw.wd-strlen(ttl)) / 2, 1, ttl);
}

/* ------------ remove a window ------------------ */
void delete_window()
{
   if (curr_wnd)   {
      if (wkw.wsave)   {
         puttext(wkw.lf,wkw.tp,wkw.rt,wkw.bt,wkw.wsave);
         free(wkw.wsave);
      }
      setmem(wdo+curr_wnd-1, sizeof (struct wn), 0);
      --curr_wnd;
      if (curr_wnd)   {
         wkw = wdo[curr_wnd-1];
         current_window();
      }
   }
}

/* ---- clear the window area and display the border ----- */
void clear_window()
{
   int height, width, y = 1;
   char line1[81], line2[81];

   height = wkw.ht;
   width = wkw.wd;
   setmem(line1 + 1, width-1, LINE);
   setmem(line2 + 1, width-1, ' ');
   *line1 = NW;
   line1[width-1] = NE;
   line1[width] = '\0';
   *line2 = SIDE;
   line2[width-1] = SIDE;
   line2[width] = '\0';
   line1[width] = line2[width] = '\0';
   writeline(1, y++, line1);
   while (height--)
      writeline(1, y++, line2);
   *line1 = SW;
   line1[width-1] = SE;
   writeline(1, y, line1);
}

/* --------- scroll the window. d: 1 = up, 0 = dn ---------- */
void scroll_window(d)
{
   if (_video.snow == 0)
      movetext(wkw.lf+1, wkw.tp+1+d,
           wkw.rt-1, wkw.bt-2+d,
           wkw.lf+1, wkw.tp+2-d);
   else   {
      rg.h.ah = d ? 6 : 7;
      rg.h.al = 1;
      rg.h.bh = _video.attribute;
      rg.h.cl = wkw.lf;
      rg.h.ch = wkw.tp;
      rg.h.dl = wkw.rt-2;
      rg.h.dh = wkw.bt-2;
      int86(16, &rg, &rg);
   }
}

/* ---------- display text in a window --------------- */
```

```
   else
      wkw.wy = ln - wkw.wtop + 1;
   while (TRUE)   {
      lineno = wkw.wtop + wkw.wy - 1;
      ptop = wkw.wtop;
      dline(lineno, foreg, backg);
      if (wkw.wx == 0)
         hidecursor();
      else
         gotoxy(wkw.wx, wkw.wy+1);
      c = getkey();
      if (c == '\r' || c == ESC)
         break;
      switch (c)   {
         case CTRL_HOME:
            firstline();
            break;
         case CTRL_END:
            lastline();
            break;
         case PGUP:
            wkw.wtop -= wkw.ht;
            if (wkw.wtop < 1)
               wkw.wtop = 1;
            break;
         case PGDN:
            wkw.wtop += wkw.ht;
            if (wkw.wtop > wkw.wlines - (wkw.ht-1))   {
               wkw.wtop = wkw.wlines - (wkw.ht-1);
               if (wkw.wtop < 1)
                  wkw.wtop = 1;
            }
            break;
         case UP:
            upline();
            break;
         case DN:
            downline();
            break;
         default:
            if (!editing && wkw.wlines <= wkw.ht)   {
               if (c == HOME)   {
                  firstline();
                  break;
               }
               if (c == END)   {
                  lastline();
                  break;
               }
            }
            if (func)   {
               frtn = (*func)(c, lineno);
               if (frtn == ERROR)
                  putch(BELL);
               else if (frtn)   {
                  wkw.wy = frtn;
                  return frtn;
               }
               c = 0;
            }
            break;
      }
      switch (c)   {
         case HOME:
         case CTRL_HOME:
         case END:
         case CTRL_END:
         case PGUP:
         case PGDN:  if (wkw.wtop != ptop)   {
                        height = wkw.ht;
                        dln = wkw.wtop;
                        while (height-- && wkw.wtext[dln-1])
                           dline(dln++, wkw.fg, wkw.bg);
                        break;
                     }
         default:    dline(lineno, wkw.fg, wkw.bg);
                     break;
      }
   }
   return c == ESC ? 0 : lineno;
}

/* ---------- move up one line --------------- */
static void upline()
{
   if (lineno > 1)   {
      if (wkw.wy == 1)   {
         if (wkw.wtop > 1)   {
            --wkw.wtop;
            scroll_window(0);
         }
void text_window(char *txt[], int ln)
{
   int height = wkw.ht;

   wkw.wtext = txt;
   wkw.wtop = ln;
   wkw.wy = 1;
   while (height-- && txt[ln-1])
      dline(ln++, wkw.fg, wkw.bg);
   wkw.wlines = 0;
   while (*txt++)
      wkw.wlines++;
}

static int lineno;
/* -------- page and scroll through a window of text -------- */
int
select_window(int ln,int foreg,int backg,int (*func)(int,int))
{
   int c = 0;
   int frtn;
   int height, dln, ptop;

   if (ln > wkw.wtop + wkw.ht-1 || ln < wkw.wtop)
      text_window(wkw.wtext, ln);
      }
      else
         --wkw.wy;
   }
   else if (wkw.wlines <= wkw.ht)
      lastline();
}

/* ---------- move down one line ------------- */
static void downline()
{
   if (lineno < wkw.wlines)   {
      if (wkw.wy == wkw.ht)   {
         scroll_window(1);
         wkw.wtop++;
      }
      else
         wkw.wy++;
   }
   else if (wkw.wlines <= wkw.ht)
      firstline();
}

/* -------- move to the first line --------- */
static void firstline()
{
   wkw.wtop = wkw.wy = 1;
}

/* -------- move to the last line --------- */
static void lastline()
{
   wkw.wtop = wkw.wlines - (wkw.ht-1);
   if (wkw.wtop < 1)
      wkw.wtop = 1;
   wkw.wy = wkw.ht;
   if (wkw.wy > wkw.wlines)
      wkw.wy = wkw.wlines;
}

char spaces[80] =
"                                                                              ";

/* ------- display a line of text, highlight or normal ------ */
static void dline(ln, foreg, backg)
{
   if (foreg || backg)   {
      textcolor(foreg);
      textbackground(backg);
      --ln;
      writeline(2, ln-wkw.wtop+3, *(wkw.wtext + ln));
      if (strlen(*(wkw.wtext + ln)) < wkw.wd-2)
         writeline(2+strlen(*(wkw.wtext + ln)),
                  ln-wkw.wtop+3,
                  spaces + 79 - wkw.wd +
                  strlen(*(wkw.wtext + ln)) + 2 );
   }
}

/* --------- write a line of text to video window ----------- */
void writeline(int x, int y, char *str)
{
   int cl[80], *cp = cl;

   while (*str)
      *cp++ = (*str++ & 255) | (_video.attribute << 8);
      __vram(__vptr(x+wkw.lf-1,y+wkw.tp-1),MK_FP(_DS,cl),cp-cl);
}

/* ------- use BIOS to hide the cursor ---------- */
void hidecursor()
{
   rg.h.ah = 2;
   rg.x.dx = 0x1900;
   rg.h.bh = 0;
   int86(0x10, &rg, &rg);
}

/* ---------- use BIOS to set the cursor type ------------- */
void set_cursor_type(unsigned t)
{
```

```
    rg.x.ax = 0x0100;
    rg.x.bx = 0;
    rg.x.cx = t;
    int86(0x10, &rg, &rg);
}
/* ----------- use BIOS to clear the screen ---------------- */
void clear_screen()
{
    window(1,1,80,25);
    gotoxy(1,1);
    rg.h.al = ' ';
    rg.h.ah = 9;
    rg.x.bx = 7;
    rg.x.cx = 2000;
    int86(0x10, &rg, &rg);
}

void (*helpfunc)(void);
int helpkey;

/* ------------- read the keyboard ---------------- */
int getkey()
{
    int c;

    if ((c = getch()) == 0)
        c = getch() | 128;
    if (c == helpkey && helpfunc)   {
        (*helpfunc)();
        c = getkey();
    }
    return c;
}

/* ------- write an error message ------------- */
void error_message(char *ermsg)
{
    int lf = (80-strlen(ermsg)+2)/2;
    int rt = lf+max(strlen(ermsg)+2,15);
    establish_window(lf, 11, rt, 14, ERRORFG, ERRORBG, TRUE);
    gotoxy(2,2);
    cputs(ermsg);
    gotoxy(2,3);
    cputs("(Press [Esc])");
    hidecursor();
    do
        putch(BELL);
    while (getkey() != ESC);
    delete_window();
}
```

End Listings

Eating An Elephant: The Project Begins



by Al Stevens

This month we inaugurate the "C Programming" column project, a running feature in which we will develop a communications program that should take several months to complete. To begin the project, we must consider the program's requirements. In my opinion two procedures are necessary during the development of a computer system: first is the requirements analysis; second is the concept of incremental implementation.

A requirements analysis results in a statement of what the program must do and how it must perform. It is expressed in a language understood by both programmers and users. Without it, you never know when you are done. Such an analysis addresses two areas: the functional requirements and the performance requirements. We will postpone the functional requirements step until later and concentrate on the performance requirements for now.

Incremental implementation means that you build a system a little bit at a time and you deliver it to the user in small increments. Subsequent increments will benefit from the experiences that precede them, not only from your experiences but from those of the user as well. Most government projects ignore this wisdom, preferring to have you automate the entire Western Hemisphere, get it done in short order, and then turn it on all at once. There are exceptions. Jim Towles, a manager of construction and facilities engineers at Kennedy Space Center, knows better and says so. "If you want to eat an elephant, you don't do it in one bite." He won't allow any of those computer folks to shove a big system at him all at once. Jim prefers smaller bites of an elephant.

Our project will take the incremental approach. First we will develop the tools to support the performance requirements. Then we will build the minimum system around those tools. The development and use of that first bite will tell us how to proceed with subsequent ones.

Performance Requirements

The performance of the "C Programming" column program must follow these guidelines:

- It must be an on-line, interactive PC program (is there any other kind?).
- It will use pop-down menus.
- It will have context-sensitive help windows.
- Data entry will be supported by a general purpose window-oriented entry library.
- There will be a general-purpose text editor package for all text data entry.
- The communications portion of the program will assume a Hayes-compatible modem.
- Xmodem data-transfer protocols will be supported (maybe Kermit as well in a later increment).
- All window functions (data entry, editor, help, menus) will use a common video window library.

For those who do not know what a video window is, here is a brief description. A video window is a rectangular area of the screen that has a border, a title sometimes, and text displays within its border. Windows pop up on top of one another. When a window pops down (goes away), it disappears, and the video displays—other windows, perhaps—that were under it are visible again.

The Window Library

To begin the C column project, we will start with the window library that supports the rest of the program. This package is at the bottom of our design. We are creating the program from the bottom up in this effort. This is not always the right direction, but we have an advantage here because our requirements fall in line with software that I have published elsewhere. Those of *you who have* read my books will recognize the similarities. This project will use a subset of that software modified for the needs of this program. The window library will be general enough that you can use it for most other window-oriented applications, but it will lack many of the slick features found in the books and in many commercial window libraries. I have left them out because they are not needed here. We want the most efficient program possible.

The window functions use the Turbo C text-mode console functions to manage window placement and data displays. For this reason you will need Turbo C, Version 1.5 or later.

Listing One, page 112, is window.h, which contains the global function prototypes, some *#define* statements, window structures, and the configuration information. A word about configuration: Many programs come

(continued from page 117)

with an installation program that guides you through selections for your screen colors and such. We are not going to include such a feature because this is a program for programmers, to be built by programmers, and to be used by programmers. Programmers can handle configuration parameters by changing the source code and compiling. To ease that process, I will use *#define* macros for configuration items wherever possible. You will see a block of such macros at the bottom of Listing One. These macros allow you to configure the program's screen colors. There are eight screen items that can be customized. These will give you a clue as to what is coming in the window package in the coming months. The eight items are: data display screens, data blocks (for example, marked blocks in a text editor), help windows, menus, the menu selector bars, data entry windows, the fields in data entry windows, and error messages. As published here, the configuration uses a pattern of blacks and whites for all items. You can use other colors if you wish. The global symbols for colors are provided in Turbo C's conio.h. The possible global symbols are BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHT GRAY, DARK GRAY, LIGHT BLUE, LIGHT GREEN, LIGHT CYAN, LIGHT RED, LIGHT MAGENTA, YELLOW, and WHITE.

Notice the esoteric prototypes and structure under the comment "internal Turbo C stuff." These provide access to Turbo C's internal video logic and are compatible with Turbo C, Version 1.5. If Borland changes these constructs in a future version, we will need to make adjustments. I have used these constructs to do direct screen reads and writes outside the realm of what is allowed by the text video functions in Turbo C. They free us from assembly language and video retrace status registers. This is highly questionable, nonportable, risky, hacker-mentality programming. You are hereby admonished never to use such practices in your own code. I love it.

Listing Two, page 112, is window.c, the window function library. The window library provides six basic functions to support windows. The window concept provides that the most recently established window is addressed by any subsequent window operations, so that when a window is deleted, the one that was established before it becomes the current window. Here is a description of each of the functions.

establish_window—Call this function to establish and display a window. It expects seven integers in its parameter list. The first four parameters identify the window's screen position in character coordinates relative to one. The four coordinates are left, top, right, and bottom. The largest possible window is, therefore, 1,1,80,25. The next two parameters give the window's foreground (text) and background colors. The color global values named above will work here. The last parameter is TRUE or FALSE to tell the window functions whether or not they should save and restore the video memory under the window. This parameter says whether the window is a pop-up window or not. It is provided as a convenience to save heap space when windows do not need to preserve what they cover.

window_title—This function will display a title in the center of the top border of the current window. Pass it the address of a string that has the title you want displayed.

clear_window—This function clears the current window and re-displays its border. If you use it, you will need to rewrite the title with *window_title*.

delete_window—This function closes the current window. If another window was established immediately before the current one, that other window becomes current. If the window was established with the last parameter to *establish_window* set to a TRUE value, then the window is erased and replaced by whatever was under it. Otherwise the window remains visible, although the window functions have no further record of it.

text_window—This function displays lines of text in the current window from an array of character pointers. Each pointer points to a string and the last pointer contains a NULL pointer value. You pass the address of the array (a pointer to a character pointer) to the function. The second parameter is an integer subscript relative to one that tells the function which entry in the array will be displayed in the top line of the window. If the array has more lines than will fit in the window, the display stops when the window is filled.

select_window—This function is used to make menus and data selectors out of windows. It assumes that you have already established the window and written the selections into it by using *text_window*. You pass it an integer relative to one that says which selection is highlighted when the menu is first displayed. The next two integer parameters specify the foreground and background colors of the highlight bar. The last parameter is a function pointer. When the function is called, it displays the menu and lets the user page and scroll up and down with the page and arrow keys. The user may also make a selection by pressing the Enter key. When Enter is pressed, the function returns the number, relative to one, of the selection. If the Esc key is pressed, the function returns zero. If the user presses a key not intended for paging, scrolling, or selecting, the function in the function pointer parameter is called (unless the pointer is NULL, in which case no call is made). This provides for help keys, function key selections, and so on. The menu functions to be added later will use this feature.

Window.c includes several other general-purpose functions that will be used throughout the program and have general application to any other program you might develop with this library. Here are descriptions of those functions:

error_message—This is a utility error message function that displays an error message in a window, beeps, and waits for the user to press the Esc key to acknowledge the error. Pass it a pointer to a null-terminated message.

hidecursor—The Turbo C window functions leave the cursor in the window that is in use. This function uses BIOS to hide the cursor for those times when its display would distract the user's attention.

clear_screen—What else? This function clears the screen.

get_key—This function reads a keyboard character. It translates function keys into the equivalent of their scan code with the most significant bit set. These values correspond to those defined in *window.h*. The *getkey* function also watches for a help function key if one is programmed and calls the help function if one is provided. These features will be explained in a later column when we install the context-sensitive help window functions.

scroll_window—This function needs some explanation. It is called to scroll the current window up or down one line. It selects one of two ways to do this on the basis of the *_video.snow* variable. The Turbo C text library has very nicely determined if the screen will display video snow when direct video reads and writes are done. It makes this decision by testing for an Enhanced Graphics Adaptor, which does not snow, a Monochrome Display Adaptor, which does not, or a Color Graphics Adaptor, which does. It goes further and decides that the Compaq version of the CGA does not snow, which, of course, it does not. The result of its decision is recorded in *_video.snow*. It will decide that some machines snow when they do not. The Toshiba T1000 laptop is an example. All this is fine, but when we use the text functions of Turbo C to scroll the screen on a snowy machine, the scrolling is too slow. This is because every character that is read or written while the scrolling text is moved must wait for video retrace to occur. So, if we allow Turbo C to scroll for us, performance suffers. For that reason, we use BIOS video services to scroll a snowy screen. You might ask why we don't just always do that. Ah, the vagaries of an imperfect world! When BIOS scrolls, it uses an annoying blanking of the screen. This annoyance is preferable to the snow or the slow, so it is accepted as the lesser of three evils.

In the next several months we'll add the help functions, menu manager, data entry screens, and a text editor. After that we'll get into the communications part of the program and its ultimate purpose.

C Crotchet Number 3: How C is Taught

(A reminder: crotchets are things that irk people. See last month's "C Programming" for an explanation and crotchets number 1 and number 2.)

My best friend and companion, Judy, is enrolled in a computer science program at a community college in Virginia. She just completed an introductory class in C. The instructor admitted that he was new to C and gave assignments for the development of a small program. He provided examples of how certain things were to be coded. Here is a fragment.

```
while (fgets(line[lineno++], 100, fp));
```

The class was told that this example is how you read a file of text into memory. Notice particularly the semicolon. Most of us can figure out what's happening here. The *while* is being used to loop until all the lines of a file are read into an array. So what's wrong with this picture? Plenty.

1. There is no boundary checking. If the file has more lines than the array, the program will probably crash.
2. The semicolon is on the wrong line. It should be indented below the *while* statement to tell the reader that it is an intended *null* statement rather than the accident it appears to be.
3. The instructor never explained the notion of an operational statement as a component of a conditional expression within a *while*, nor did he address the use of the *null* statement, which is there merely to give the *while* something to do until its condition is FALSE.
4. The idea that the statement that reads the file also returns the end-of-file condition and can be tested when it is executed was not explained.
5. The level at which the class is offered suggests that the students are not ready for such concepts as auto-increments that occur at the same time the subscripting integer is used as a subscript.
6. The constant 100 is a danger spot. If *line* is a two-dimensional array of strings, the constant can be coded with the *sizeof* operator. At least the constant should be equated to a global symbol for better reading and easier maintenance.

Here is a clearer presentation of the same logic with boundary checking added.

```
while (lineno < MAXLINES) {
  rtn = fgets(line[lineno], LENGTH, fp);
  if (rtn == NULL)
    break;
  lineno++;
}
```

Sure, it uses several lines where one will do, and sure, few seasoned C programmers really write code like that. This crotchet is not for expert C programmers who only program. They are forever encouraged to use the language to its fullest. This crotchet is for those of you who would teach. C encourages tight and concise expressions and for that we hold it dear. The new C programmer, however, needs to move carefully and slowly into such advanced usage. This freedom of syntax is the chief object of criticism of C by devotees of other languages but educators do not need to teach it so. Programmers can learn nice, readable statement sequences like the one above and approach the tighter, more elegant side of the language at their own pace. What the students were taught was that the instructor's example would read the text into the array. They never learned why.

I plan to devote a future column to the problems of teaching C.

C Crotchet Number 4: Grousing about Upgrade Fees

I like Turbo C. And someday when I get a new corrected version, I might begin to like QuickC. While doing research for books on these compilers, I spent a lot of time on the CompuServe and BIX on-line services. There is no better way to clear up a compiler or language problem than by using the related forums of these services. Lately a common theme has been running through the discussions, one that wastes connect

time and deserves comment. When Borland or another vendor releases a new version of a C compiler, they usually charge a nominal upgrade fee to registered users. Such a fee seems reasonable when you consider the value of a C compiler. Many of you will remember what we used to pay for C compilers that had far fewer features. The common complaint, however, is that the programmers feel ripped off by the fee. They argue that since the upgrade includes the correction of known bugs, the vendor should provide it for free. We all know that those multipage full color ads (do you believe Turboman?) and those thousands of programmer person-hours are not free to the compiler vendors. The revenue to continue promoting and improving the product has to come from somewhere. Yet the grumbles persist. One such forum conversation went on for several days about a $10 upgrade charge.

My pal Bill Chaney gets hot and says such folks are "so tight they wouldn't spend a nickel to see a stink bug eat a bale of hay." Easy, Bill ....

The C Programming Library

The book to read this month is *C Programmer's Guide to Serial Communications* by Joe Campbell (Howard W. Sams & Company, 1987). There are over 650 pages of text and code dealing with the subject of asynchronous serial communications on microcomputers. Campbell presents the subject matter in clear but advanced language. This is no book for beginners. You need to be a C programmer at the very least, and it helps to already have a passing acquaintance with the murky depths of serial communications.

Campbell's writing style is refreshing and makes for good if not light reading. He likes to use words that will send many of us scrambling for Webster's, and he makes no attempt to conceal his contempt for what he considers an inferior design. His treatment of the Xmodem and Kermit data-transmission protocols leave the reader with the impression that in his view the world would be a far better place if Joe Campbell had been allowed to design them. Mostly his criticisms are valid and to the point even when obviously born from the precious perspective of clear hindsight, but if I were Ward Christensen or a few others my ears would be burning. In the preface to the book, Campbell refers to his "humble recognition of the demands of [the subject of data communications]," and that is the last evidence of humility you'll find in this book. An occasional lapse in humility, however, is OK when the apparent arrogance is justified or substantiated by equally apparent intelligence and expert knowledge, and Campbell delivers in this book. His description of Kermit is the first one I've read that didn't require at least a second reading to understand.

The book discusses communications theory, hardware, software, and the C language implementations of these concepts. It is a healthy treatment of these subjects and a necessary addition to the library of those involved in C projects where computers talk to one another through serial interfaces. Recently I downloaded a C source code archive file that purports to implement the Kermit protocol in a Unix environment. I wanted to port the undocumented code to the PC and learn from it. To my chagrin I found that the one source file in the archive is incomplete, abruptly ending in the middle of a function. With the explanations in Campbell's book I can now attempt to provide the missing code. Campbell's book does have C functions that implement the Xmodem protocol, but does not have equivalent code for Kermit. Drat.

Until I read this book I never fully understood why serial input-output and the RS-232 "standard" had been so confused for so many years. When I was consulting in 1978, the microcomputer was new to the business world and plug-and-go appliance computer systems were rare. I earned a substantial part of my living with a break-out box, some cable stock, DB-25 connectors, and a soldering iron. Every installation had some new serial printer that needed to be connected to some hacked-together microcomputer and nothing ever fit. In the chapter on RS-232 control Campbell explains that RS-232 was never intended to be used for printer handshaking. How come I never knew that? The original RS-232 intentions notwithstanding, most printers back then could be bought with a serial port. This option allowed users of smart terminals and modems to slave a printer to the auxiliary serial port on their terminal. As a consequence, the designers of early microcomputers put serial printer interfaces in their machines to accept those printers already in use. The Centronics parallel standard for printers was around but few systems used it. Maybe the UARTS and line driver chips were cheaper or more available than parallel drivers. Whatever the reason for the neglect of the Centronics interface, when IBM adopted it for the PC they started a trend that eventually sent me happily out of the 1200 baud connectivity business. I could have used Campbell's book back then.

Campbell earns my respect for his acknowledgment of Cole Porter and Johnny Mercer, two popular music composers from the pre-MTV era. They certainly weren't programmers and probably never heard of C, RS-232, or even MIDI, but their legacy to our culture is appreciated among those of us who dawdle too long and too late in the silent ones and zeros.

Next month we will add some features to the window library, look at another book, and kick some more crotchets around. Until then, stay fit and keep coding.

Availability

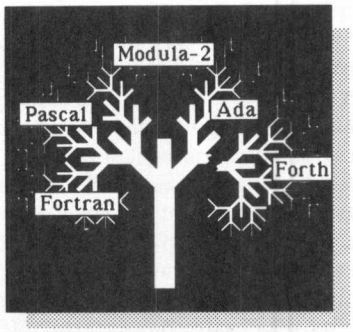
All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr. Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 112.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

What is Modula-2?



by Kent Porter

Introduced in 1982, Modula-2 is a relative newcomer that has only lately begun to gather a following. The evidence of its rising importance is the recent introduction of several new Modula-2 compilers for the PC; there's a comparative review elsewhere in this issue. This review discusses specific implementations. Here we'll discuss the language as a whole.

At first glance, Modula-2 looks a lot like Pascal. Don't let appearances deceive you. Both languages were sired by the venerable Niklaus Wirth, so they share genetic similarities. While Modula-2 is a new language, every bit as powerful as C — perhaps it's even more so in some ways.

To describe Modula-2 as the successor to C would be tantamount to picking a fistfight with most readers of this magazine. Certainly it's like cussing in church. So let me hasten to state that I'm not advocating Modula-2 or predicting the demise of C, which still has plenty of good lines of code left in it. Personally, I like C and I've written a lot of stuff in it. But I do believe that Modula-2 is an alternative to C and that it is a language well worth learning and using.

You might say that Modula-2 combines Pascal, without its defects, and C in a more readable format, plus extensions. Wirth developed Modula-2 as a language for writing operating systems, utilities, and the applications that run under them. Thus, unlike Pascal, which was designed as a teaching language, Modula-2 rests solidly on a foundation of real-world software requirements.

Because Modula-2 is an evolutionary outgrowth of Pascal and an alternative to C, it's instructive to compare it with both, and that's what we'll do. This isn't a comprehensive treatment, but it covers most of the main differences.

First, Modula-2's similarities to Pascal: The definition of data structures, user-defined types, constants, and variables in Modula-2 is virtually the same as in Pascal, with some extensions mentioned later in this article. The assignment statement (*a := b*) is the same, as are the use of semicolons and the ability to pass parameters either by value or by reference. The latter two are very close to C conventions as well.

Semantic Differences

On a superficial level, one of the first things you notice about Modula-2 is that it's case-sensitive like C and unlike Pascal. Whether or not that's an improvement is debatable; doesn't make much sense to me, but that's how it is. All Modula-2 keywords are uppercase. By stylistic convention, user-assigned identifiers are lowercase, but can include optional capitals. Case sensitivity means that, for example, *readkey*, *readKey*, and *ReadKey* are three different identifiers.

Modula-2 offers the same data types as Pascal, plus two new ones: *CARDINAL* and *BITSET*. A *CARDINAL* is an unsigned integer and a *BITSET* is a machine word whose bits are individually manipulated with unique operators. These are C-type additions that have no counterparts in standard Pascal.

Commercial Pascal compilers implement the nonstandard string type *STRING [n]*, which is usually an *ARRAY [1..n] OF CHAR* with a length byte in the 0th element. Instead Modula-2 implements this string type as an *ARRAY [0..n] OF CHAR* with a null terminator, which is exactly the same string type that C has.

Pointer declarations are more intuitive in Modula-2 than in either Pascal or C. Here is an example:

```
VAR intPtr : POINTER TO INTEGER;
```

Modula-2 lets you declare the absolute address of a variable using the format (in the PC)

```
VAR absVar [segment:offset] : <type>
```

This format is similar to Turbo Pascal, but is generally not available in other Pascals or in C. This is extremely useful in system programming — for example, when you need to refer to fixed-location system variables such as the ROM BIOS data area.

Both C and Pascal are inconsistent with regard to the application of block statement delimiters (*BEGIN/END* in Pascal and paired curly braces in C). Constructs such as *IF*, *WHILE*, and *FOR* can control both single statements and statement blocks; for a single statement, delimiters are not used; otherwise, they are. Thus in Pascal, we have

```
If a = b then
   c := d;

but

If a > b then
Begin
   c := a;
   a := b;
   b := c
End;
```

C syntax exactly parallels these. Not so in Modula-2: all such constructs are blocks by definition. Thus, you do not need to use *BEGIN* since it's implicit in *IF*, *WHILE*, or *FOR*. The

construct always concludes with *END*, even if it only controls a single statement. Thus, in Modula-2 the cases that are analogous to the ones above are as follows:

```
IF a = b THEN
   c := d
END;
```

and

```
IF a > b THEN
   c := a;
   a := b;
   b := c
END;
```

This consistency makes it easier for beginners, and saves everyone the trouble of having to run down unbalanced block delimiters.

The Pascal loop constructs carry into Modula-2: *REPEAT . . . UNTIL*, *FOR*, and *WHILE*. The latter two have direct counterparts in C, and *REPEAT . . . UNTIL* is similar to (but the opposite terminating condition of) C's *do . . . while*. Additionally, Modula-2 offers an open-loop structure using the *LOOP* and *EXIT* keywords. *LOOP* begins an iterative block terminated by *END*; *EXIT* transfers control beyond *END*. This example shifts a string to uppercase:

```
i := 0;
LOOP
   IF strng [i] = CHAR (0) THEN
      EXIT
   ELSE
      strng [i] := toupper (strng [i])
   END; (* of IF *)
   INC (i)
END; (* of LOOP *)
```

Note that *EXIT* skips over intervening statements, such as the incrementing of *i*.

Modula-2's CASE statement is better than Pascal's because *BEGIN/END* are not needed to delimit a multistatement selection, and because there is an *ELSE* condition of last resort. The *ELSE* condition is usually provided by commercial Pascal compilers, but this situation is outside the standard one. To eliminate *BEGIN/END*, Modula-2 employs an *OR*-bar in lieu of a semicolon (or a break statement in C) to indicate the end of a selection. Here is an example:

```
CASE x OF
   1:       A;
            B |
   2:       B |
   3..7:    C
ELSE
 IllegalCase (x)
END; (* of CASE *)
```

The Modula-2 *CASE* statement is still not as flexible as C's *switch( )*, though. *CASE* doesn't provide for flow from one case selection to another. In C, you could write this as

```
switch (x) {
    case 1:     A ( );
   case 2:      B ( );
                break;
   ...
} /* end of switch( ) */
```

Here, when *x* is 2, the program does *B* only; when *x* is 1, it does *A* and then *B*, flowing from *case 1* to *case 2*. In effect, the label for *case 2* is in the middle of the sequence for *case 1*, and that's legal in C. It's not in Modula-2.

In Pascal, subprograms can be functions or procedures. A procedure has some effect external to itself, whereas a function returns a value assignable to a variable or that is eligible for output. C has only functions, although they might be procedural, value-returning, or both; a Pascal function can also do both. Modula-2 brings symmetry to the party by providing only procedures. The nature of a Modula-2 procedure is identified by its heading, just as it is in C, but the declaration is more Pascal-like. The heading

```
PROCEDURE hypSin (theta : REAL) :
                                REAL;
```

defines a function that returns the hyperbolic sine of theta as a REAL. Conversely, the heading

```
PROCEDURE upShift (VAR strng :
                        ARRAY OF CHAR);
```

declares a procedure that presumably shifts its parametric string to upper-case. The Modula-2 calls are just as they would be in the corresponding C or Pascal incarnations:

```
hsin := hypSin (angle);
upShift (aString);
```

Like a Pascal or C subprogram, a functional procedure in Modula-2 can both return a value and have an effect external to itself.

But Modula-2 adds a couple of wrinkles that will take programmers in both camps by surprise. The first is a *RETURN* statement. Pascal doesn't have one. C does, and Modula-2's is identical to it. An unadorned *RETURN* anywhere within the procedure body forces a return without value, which is legal inside any purely procedural subprogram. You can use the statement

```
RETURN x;
```

in a functional procedure to send a value back to the caller; *x* can be a local variable or an expression.

The second surprise in Modula-2 is that the *END* statement that closes a subprogram must include the procedure's name. Thus, the outline of *hypSin* in Pascal might be

```
FUNCTION hypSin (theta : REAL) :
REAL;
VAR result : REAL;
BEGIN
   (* Compute result, then . . . *)
   hypSin := result
END;
```

In Modula-2, the same outline is written as

```
PROCEDURE hypSin (theta : REAL)
                              : REAL;
VAR result : REAL;
BEGIN
 (* Compute result, then . . . *)
 RETURN result;
END hypSin;
```

If you check back to the declaration of *upShift* and you'll discover that one of the long-discussed defects of Pascal has been removed from Modula-2; the parametric string has no stated subscript range. Because of its strict type-checking, Pascal has never been very smart about letting subroutines accept arrays of

varying sizes. Microsoft Pascal has a thing called a superarray and Digital Research's Pascal MT+ honors conformant arrays, but these arrays are implementation-specific stabs at getting around standard Pascal's hidebound stupidity when it comes to array-handling in subprograms. In general, if you want to process a three-element array of some type in Pascal, you need a subroutine that is different from one that you need to process a four-element array of the same type in an identical manner. C is a little smarter: you can pass a second argument indicating the number of elements.

Modula-2 is a lot brighter in this regard than both of the other languages. Its procedures accept a parametric array of any size. The only restrictions are that the array must be one-dimensional and of the declared data type. Within the body of the program, the standard function *HIGH (arrayname)* yields the upper-bounding subscript of the passed array. Thus, you might write *upShift* as

```
PROCEDURE upShift
        (VAR strng : ARRAY OF CHAR);
VAR i : CARDINAL;
BEGIN
  FOR i := 0 TO HIGH (strng) DO
    strng [i] := toupper (strng [i]);
  END (* of FOR *)
END upShift;
```

You can do similar operations involving *HIGH* on arrays of any other data type, including user-defined types such as arrays of structures. Modula-2 makes it easier to deal with arrays in subroutines.

I/O Handling

Despite the syntactic similarities, Modula-2 is more like C than Pascal. The 40-word core language of Modula-2 deals chiefly with declarations, definitions, expressions, and import/export lists (discussed later). In other words, the defined language is primarily concerned with internal matters and does not address I/O. For I/O, it relies on external libraries. This reliance is C-like and very different from Pascal.

Pascal comes with built-in I/O procedures. The main ones are *WRITE(LN)* and *READ(LN)*. These procedures are remarkably supple routines, capable of communicating with the console, disk files, and auxiliary devices such as the printer and those attached to the serial port. The C standard library contains counterparts, which are the *printf( )* and *scanf( )* families.

Architecturally, there are two fundamental differences between Modula-2's *WRITE* and C's *printf( )*. First, *WRITE* is built into Pascal, whereas *printf( )* comes from an external C library. Second, *WRITE* encompasses virtually all output, whereas the version of *printf( )* (*fprintf( )*, *sprintf( )*, and so on) that the C programmer selects depends on the output destination.

Yet, they share one similarity: in both cases, they must evaluate the data types of a variable number of parameters, and must act on those types as appropriate. In one instance, you might tell the routines to output a string, then an integer, then another string, then a *REAL* (double to you C guys) to a file. In the next instance, the types and order might be entirely different, as well as the destination.

This flexibility is great from the programmer's viewpoint because it limits the number of output alternatives. Just pick the call you need, set up the parameters, and away you go.

But it has penalties in terms of .EXE program size and efficiency. In both Pascal and C, the output functions must be able to pick apart the parameters and dispatch subtasks to handle them. This additional work all adds up to overhead.

Modula-2 takes a radically different approach. Each I/O routine is bound to a specific data type. Thus, there are different library I/O routines for integers, reals, cardinals, strings, and so on. There's even a separate routine (*WriteLn*) to advance to the next output line.

Say you want to write the output

```
This program executed 10 times
                                    <CR>
```

In C, you might write the statement

```
printf
 ('This program executed %d
                        times\n', ntimes);
```

The corresponding statement in Pascal is

```
WRITELN ('This program executed ',
                    ntimes, ' times');
```

Modula-2 requires the following:

```
WriteString ('This program executed ');
WriteInt (ntimes);
WriteString (' times');
WriteLn;
```

This hardly seems an improvement at first glance. One source line in either C or Pascal becomes four in Modula-2. But consider the implications. No Modula-2 I/O routine needs to interpret its parameter list; it merely acts on a specific, predictable data type.

The net result is a reduction in executable complexity and, thus, overhead. Modula-2 *trades more* verbose source language for more efficient programs. The distinction is subtle and hard to accept at first, but it pays off in the long run.

Modula-2 also eliminates one of Pascal's bad ideas: the typed file. According to Pascal doctrine, any

given disk file consists entirely of one data type: byte, real, user-defined, or whatever. It doesn't work this way in practice, though. Many software systems require a variable-length file header consisting of several different control structures, followed by a series of fixed-length data records. You can convince Pascal to work with such files (see *DDJ*, May 1988, page 92), but the solution is ugly. That's one reason why C is preferred over Pascal for many applications: C doesn't require typed files.

Neither does Modula-2. Like C, it provides the generic typed object *File*, and it doesn't care what you read or write once you open the file. Modula-2 implementations generally furnish separate routines for text and binary file I/O, including data structures, random access, and file-pointer positioning. Consequently, Modula-2 is as capable as C at manipulating files.

Modularity

The name *Modula-2* comes from the language's emphasis on program modules at both the source and compiled levels. Pascal borrows this concept when it furnishes units consisting of interface files and implementation files. A rough analogy also exists in C, when an .H file contains function prototypes and other declarations involving a library. But neither Pascal nor C carries the modular idea as far as Modula-2 does.

Modula has three kinds of modules:

- Definition — which is a list of constants, types, variables, and procedures exported by an associated library. This module is the library's interface, and describes what is available from the library.
- Implementation — which is code that implements the elements shown in the definition module.
- Program — which is the highest-level portion of a program, containing the main body and supporting elements.

Here is an example that illustrates how these pieces fit together. Let's say you've developed a set of general-purpose routines for managing the user interface. You can call these routines from any program, so you put them into a module named *USERINT.MOD*. The compiler, finding that *USERINT.MOD* begins with the keyword phrase *IMPLEMENTATION MODULE*, arranges the result into library format.

Because you must describe what the *USERINT* library contains, you write a companion file called *USERINT.DEF*. This file opens with the identifier *DEFINITION MODULE* and consists of three main parts. The first part is an optional *EXPORT QUALIFIED* list, which gives the names of all identifiers from the *USERINT* library that are externally visible. This list is the same as a list of *PUBLIC* statements in assembly language. Because the export list just shows names, it's necessary to explain what they are, which leads to the remaining two parts of the definition module. One part declares the public constants, types, and variables in the usual fashion. The other part lists the headings of exported procedures.

In most Modula-2 implementations, you must also compile the definition module. The output is a symbol file — usually with a name such as *USERINT.SYM* — that contains the identifier names and associated binary-encoded control information. The source listing serves as a reference for programmers who want to use the library.

Now, the *USERINT* library is available to other programs. Suppose you write an accounting package called *BOOKKEEP*, which contains the main body of the application and some local procedures and variables. The source program is called *BOOKKEEP.MOD*, and begins with the keyword *MODULE*.

Modula-2 requires that you notify it of the libraries and their associated identifiers that you intend to use. Consequently, the heading of *BOOKKEEP.MOD* begins with one or more import lists. In the case of *USERINT*, you write

```
FROM USERINT IMPORT
<comma-delimited list of identifiers>;
```

if you want to use some of the symbols. Alternatively if you want access to everything from *USERINT*, you can alternatively write

```
IMPORT USERINT;
```

When the compiler encounters an import list, it opens the named file, which is the compiled definition module in this case, and looks up the imported identifiers, incorporating them into its symbol tables. In some implementations that have proprietary linkers, the compiler also builds a list of libraries that it will search to resolve external references at link time.

Although this profusion of modules seems complex, it offers a number of advantages. One is that a program can use imported identifiers as though they were its own. As an example, if *USERINT* exports a type called *PopWindow*, the importing program can declare variables of type PopWindow. Similarly, the user program can simply refer to variables and call routines exported from *USERINT* without further qualification.

Another advantage of this approach is implementation hiding. The definition module shows what is externally visible in a library, without divulging how the library actually does its thing. This is important when distributing commercial libraries for sale, and also in group projects. If you decide to market your *USERINT* library, I can buy the object form of the implementation, plus the source for the definition. After listing the definition file as a reference, I compile it into a form understandable to my compiler. Then I simply use the library, without having detailed knowledge of its inner workings.

An offshoot of implementation hiding is a unique feature of Modula-2 called an opaque type. This type is a data type defined in the implementation but not in the definition module, which simply exports it by name without elaboration. An opaque type is usually a record type whose format and fields are unknown to the user program. The program can declare variables of the opaque type and pass them around, but the program can't extract from or write information into the fields. For that, the program must rely on library routines. An opaque type allows library implementors to alter the data structure

without affecting programs that use it, and also to prevent users from tinkering with field contents that might cause the library routines to go insane.

Standard Modules

Like C with its .H files describing the standard library, Modula-2 is surrounded by a number of standard modules. Most of these modules are concerned with I/O, but other modules deal with dynamic allocation, interprocess communication, transcendental functions, and so on. In the de facto standard work *Programming In Modula-2*, Niklaus Wirth specifies a core set of standard modules, including their names as well as the names and purposes of the procedures they incorporate.

Vendors of Modula-2 compilers usually ship additional libraries to enhance the functionality of their products. These might do such things as embedding hard code for an 80x87 coprocessor, providing timing routines, windowing, graphics support, mouse interface, and the like. The more libraries and procedures the product provides, the more immediately usable the package is.

It's not uncommon for two libraries to contain duplicate procedure names that do different things. For example, the standard modules *InOut* and *FileSystem* both furnish a procedure called *WriteChar*. Of course, if you use only one of these modules, it doesn't matter; the conflict only exists when you import *WriteChar* from both modules. In that case, you have to tell the compiler which *WriteChar* you mean. You do this by prefixing the source module name, using a period as the separator. Here are the code lines that refer to both versions of *WriteChar*:

```
InOut.WriteChar (ch);  (* to display *)
FileSystem.WriteChar (fyle, ch);
                              (* to file *)
```

Coroutines

Neither Pascal nor C supports concurrency directly through elements of the language, but Modula-2 does. In Modula-2 terminology, concurrent processes are called coroutines.

Normal routines observe a superior/subordinate relationship: *B* is called by *A*, which is called by the *main* program. Therefore, main is superior to *A*, and *A* is superior to *B*. *B* can never call *A* because *B* is subordinate to *A*.

Coroutines, on the other hand, are equals. If *C* and *D* are coroutines, each can transfer control to the other. *C* might run for a while and then give control to *D*, which can later switch back to *C*, and so on. The coroutines thus achieve concurrency by sharing the machine on a time-multiplexed basis.

Extending this idea a little further, you can see that *C* might be a scheduler task that dispatches a number of other coroutines (*D*, *E*, *F*, and so forth). When *D* gains control, it might run through one iteration of a loop and return the machine to *C*, which then dispatches *E*. This process continues until it comes full circle and *D* again gets control.

Each coroutine owns its own workspace where it maintains local variables, a process stack, and a state table. When a coroutine relinquishes control, it saves its context in the state table, and then sets a global pointer to indicate where it left off. The local stack and variables remain intact. When the coroutine is again activated, it reloads its registers from the state table and resumes execution. Coroutines can communicate among themselves using global variables.

The program registers a coroutine by calling the Modula-2 procedure *NEWPROCESS*. This call initializes the workspace and the global pointer associated with the coroutine. The transfer of control from *C* to *D* is simple from within source code:

```
TRANSFER (taskC, taskD);
```

In other words, once the coroutines are registered via *NEWPROCESS*, everything is done with pointers. *TRANSFER* saves C's current *IP* in the pointer called *taskC*, triggers the context switch, and then sets the *IP* register to the point of resumtion for *taskD*. *D* then runs until the next *TRANSFER* is encountered.

Coroutines are especially important in embedded controllers and operating systems, but they also have uses in end-user applications. An example is a heavily CPU-bound computation program. You can place the long-running calculation in the background by making it concurrent with a keyboard processor. Periodically, the computation gives control to its coroutine so that a keystroke, if waiting, can be processed; the handler then transfers control back to the calculation routine.

The related procedure, *IOTRANSFER*, also part of standard Modula-2, establishes a coroutine relationship between a process and an interrupt-service routine. *IOTRANSFER* says, in effect, "when this interrupt occurs, simulate a *TRANSFER* to interrupt the running process and switch to the ISR." Thus, you can rather easily achieve some level of multitasking with Modula-2 on a *DOS machine* by hooking a scheduler to timer interrupt *1Ch*.

Summing Up

This article is by no means a comprehensive description of Modula-2, but it covers the main points of the language. If you want to find out more, the definitive source is Wirth's *Programming In Modula-2* (Springer-Verlag), which is now in its third edition. For learning the language and also as a reference, an excellent text is K.N. King's *Modula-2: A Complete Guide* (Heath, 1988).

So that's what Modula-2 is, and why it's a worthy alternative to both Pascal and C.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Netwurkz and PL/D

This month I won't be presenting the object-oriented language interviews I'd planned because of a product that has been sitting on my shelf for months and that I've finally had a chance to start playing with. Writing about this package will lead me back into parallel processing and more specifically into neural networks. I guess I'll get back to object-oriented interviewing next month.

The neural network paradigm, as I've mentioned in past columns, loosely models the structure of the nervous system, with many nodes or neurons linked in a large network whose overall behavior is a function of its inputs, its topology, and properties of the neurons. It is a naturally parallel paradigm.

As I mentioned in the July issue, my friend Jürgen Fey is investigating neural net models. Having built a transputer-based system, Jürgen now wants to see what he can do with it, and neural nets seem like a natural next step, particularly given the ease with which transputers can implement a custom network of processors.

My hardware knowledge doesn't extend much beyond setting jumpers and cleaning edge connectors, so I have not followed Jürgen's example. But I really wanted to play around with neural nets, and I wanted it to be, well, easy. I was looking for some sort of simulation that would run on one of my computers without additional hardware and without my having to write the simulation myself. Obviously, such a simulation would be of no practical value: simulated parallel processing on a single-processor machine is only useful for learning, for experimenting

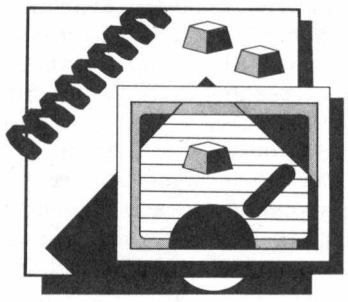
by Michael Swaine

purposes. My purposes.

Netwurkz turned out to be ideal for my purposes.

Nerl Netwurkz

Netwurkz is a neural network simulation written in a system language called PL/D. Netwurkz and PL/D are



the brainchildren of Dennis Reinhardt, president of DAIR Computer Systems in Palo Alto, Calif. Reinhardt distributes the source for both the simulation and the compiler itself at a reasonable price.

What Reinhardt has implemented in Netwurkz is the simplest kind of neural network—a deterministic, single-pass, feed-forward net. Its simplicity makes it a good tool for getting started with neural nets, and the implementation, although cumbersome, certainly gives you a sense of what it really means to build a neural net, as you laboriously cobble together your network of neurons.

The physical system that all neural network systems simulate is this: some 1,010 information-processing cells connected via input channels called dendrites and output channels called axons. Each cell typically has one axon and many dendrites. Within the cell, information flows electrochemically from dendrites to cell body to axon, with the cell applying some function to the inputs to produce the single output signal sent down the axon. Between cells, axons connect with dendrites.

Some attempts to model this system have used matrix models, tagging the connections that actually exist in a large sparse matrix that represents every possible connection.

A more efficient way to implement such a model is via list storage, letting individual lists indicate (1) the cell and (2) all the cells whose dendrites reach it: the node and its sources as head and tail of a list. Reinhardt's system language PL/D turns out to be good at list processing. Now, Reinhardt is intrigued by neural networks, and it would make a nice story to say that he created a language and a compiler in order to do neural network research, but the truth seems to be that his language just happens to be well suited to supporting the kind of data structures he needs for his neural network research.

The language also happens to be interesting in its own right.

Programming Language/Dennis

Dennis Reinhardt's real motivation for developing PL/D was a mundane observation. He noticed, not for the first time, nor was he the first to make the observation, that assembly language and high-level languages each have their strengths. He went on to get clearly in mind what those respective strengths were, at least from his perspective, and set out to develop a language that embodied the best of both levels of programming.

History does not record how many languages have been developed from exactly this motivation. History does demonstrate, though, that the same motivation can lead to rather different solutions, depending on the programmer's sense of just what the relative strengths of assembly lannguage and HLLs are and on the programmer's tastes and abilities.

Reinhardt's perspective, at least in part, was that assembly language provides tight control and HLLs provide convenient notation, and he tried to provide both in PL/D.

A simple assembly-language operation might look like:

```
LOAD    A
SUB     B
STO     C.
```

These three lines perform one meaningful operation; reflecting this notationally could be accomplished by putting them on one line:

```
LOAD A  SUB B  STO C.
```

PL/D does exactly this, with compiler-

style notation:

```
A – B —> C ;  or:
A ;
   – B ;
      –> C ;
```

(PL/D is forgiving of syntactic variations because it is, at the expression level, essentially translating a text stream into code.) At this level, PL/D is an assembler that accepts a notation for expressions more like that you'd expect from a compiler.

Above the level of the expression, PL/D provides the structures you would expect from a *HLL: IF ... THEN...ELSE...ENDIF, FOR... DO...NEXT, WHILE...THEN ...ENDWHILE,* and *CHOOSE...CASE... BUT ...ENDCH*. It is these structures that make PL/D a compiler.

PL/D has a couple of other nice features. Any declaration made within a subroutine is local to that subroutine, for more structuring. And Reinhardt has built a lot of compile-time orthogonality into the language. All four of the control structures have compile-time analogs with exactly analogous syntax, such as *FOR$... DO$...NEXT$*. You could implement the *Byte* magazine Sieve of Eratosthenes algorithm entirely in compile-time code in PL/D. PL/D allows you to control any libraries you want to include, so you can keep programs small. And it's a one-pass compiler.

OK; I'm not writing a review of PL/D here, and if I were I'd probably find some problems to gripe about. The point is that it is an interesting little language, and it provides a means for exploring simple neural networks.

The Neuron as DATA Statement

The neuron of the neural network model that Reinhardt has created in Netwurkz is implemented as a PL/D *DATA* statement. The list structure of cell and input cells becomes a *DATA* statement that lists the cells. These *DATA* statements are interpreted and executed (sequentially) by a network executive routine. In a true parallel implementation of a neural network, the cells themselves would have access to processors and could fire in parallel.

The *PL/D DATA* statement takes one of these forms:

```
DATA Name v1 ... vn ;
DATA ** v1 ... vn ;
```

Here, the first form defines *Name* as the current location counter and allocates *v1* through *vn* to the current through current + (n–1)st starting locations (actually current + 2*(n–1), but that's not important to an understanding of this use of the *DATA* statements). The second form is identical except that it does not define a name.

A neuron in a neural net model typically does something such as summing the values of its dendritic signals or summing all those that exceed some threshold, and there are many variations on this theme. Real neurons often have inhibitory dendrites, whose signals inhibit the signals of other dendrites. Reinhardt has implemented several simple neurons that do things more suited to his immediate tutorial purposes than to modeling the neural processes of the brain. The Netwurkz simulation includes the neurons *CON, EQ+, SUM,* and *MAX*.

The *CON* neuron defines a constant. I question its neural reality; it sounds suspiciously like the infamous "grandfather cell." It is used to build a database. Its syntax is:

```
DATA Name CON Value ;
```

The *EQ+* neuron counts the number of exact matches between corresponding *a* and *b* inputs. The count is saved in the location reserved by the *0*. Its syntax is:

```
DATA Name EQ+ 0 n a1 b1 ... an bn ;
```

The *SUM* neuron adds the values of its *n* inputs *a1* through *an* and stores the sum in the location reserved by the *0*. It also stores a pointer to a reference neuron. This business of providing two outputs from the cell is counter to the one-axon rule for real neurons, but Reinhardt says that the same effect could be achieved with one output (or one axon) just by time-sharing the channel between two messages. Its syntax is:

```
DATA Name SUM 0 refad n a1 ... an ;
```

The *MAX* neuron compares its two inputs, placing the larger value and the address of the neuron that provided it in its two reserved spaces—another two-axon simplification. Its syntax is:

```
DATA Name MAX 0 0 a1 a2 ;
```

With these simple neurons, Reinhardt implements an even simpler spelling checker that has a five-word vocabulary and only recognizes four-letter words. Let me try that second constraint again: It recognizes only words of four or fewer letters. Both these constraints are imposed merely to keep the example simple and easy to follow; they are easily relaxed.

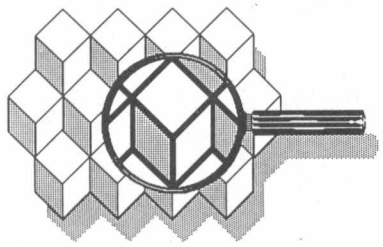
The spelling checker compares the first word in its vocabulary with each of the others to find a best match. It accepts and assigns goodness weights to shifted matches (Ike vs. Mike), elisions (Swine vs. Swaine), and transpositions (Dbob's vs. Dobb's), and it gives preferential weight to initial-letter matches. These few match rules are easy to *implement*, easy to understand, and easy to adjust the weightings for.

Other than that, the spelling checker is lacking just about everything you'd like it to have. It doesn't learn; its vocabulary is insignificant; and as it is currently implemented, adding new words to the vocabulary requires adding new neurons at every level of processing. Giving it a word to check means keying in a number of *DATA* statements. But what is nice about the Netwurkz implementation is that the model makes these deficiencies so apparent and provides a framework in which to think about how to fix them.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

OF INTEREST



Literature, Libraries, and Compilers

The long awaited second edition of *The C Programming Language* by Brian Kernighan and Dennis Ritchie is now available from **Prentice Hall**. This edition describes the new features that have been added to C over the past ten years, such as structure assignment, enumerations, and stronger type checking.

Also covered are the features of the draft ANSI Standard for C, including function prototypes, the standard library, and portability issues. Included in the book's eight chapters *and three appendices* are programs that have been tested as well as exercises to help readers test their understanding. This 261 page book sells for $28. Reader Service No. 20.
Prentice Hall Inc.
200 Old Tappan Rd.
Old Tappan, NJ 07675
201-767-5054

Elsevier Science Publishing Company has announced the release of a user-friendly guide to effective Cobol 85 programming entitled *COBOL 85 for Programmers*. Written especially for experienced Cobol programmers, this guide provides concise, detailed descriptions of the new language features and shows programmers how to combine applications of both old and new features. Donald Nelson, a principle architect of *COBOL 85*, authored the book which sells for $25. Reader Service No. 29.
Elsevier Science Publishing Co.
52 Vanderbilt Ave.
New York, NY 10017
212-370-5520

Solution Systems has recently released C-Worthy interface Library for OS/2. C-Worthy allows developers to port their code from MS-DOS to OS/2 and back again.

C-Worthy is a system for creating and managing an entire user interface including on-line help, errors, screen display, and data input. Programmers can use C-Worthy to create interfaces that resemble the Presentation Manager and Lotus 1-2-3 or interfaces that are uniquely suited to a particular application's needs.

C-Worthy contains over 350 functions as well as documentation exceeding 1,000 pages of examples, indexes, and descriptions. The product is compatible with Microsoft C, Version 5.1. Source code is available for both the MS-DOS and OS/2 versions.

C-Worthy Interface Library sells for $195, $295 with the Form Interface Library, and for $495 with Forms and Library Source. The "All Compiler" version (Lattice C, Microsoft C, Quick C, and Turbo C) for MS-DOS can be purchased for $595. Reader Service No. 21.
Solution Systems
541 Main St., Ste. 410
S Weymouth, MA 02190
617-337-6963
800-821-2492

Lattice Inc. now offers the Lattice C Compiler with features specific for creating programs to run under Digital Research's Concurrent DOS. Containing all the functions of the current Lattice C Compiler, the new C compiler also generates native mode programs which means applications created by the new C compiler will not require PC mode emulation when running under concurrent DOS operating system.

Lattice has also added library functions for file passwords and file sharing, alternative end-of-file treatment, multi-sector I/O, and shared peripherals. Additionally, an option on the compiler can generate code that can be shared to save memory when multiple copies of the program are running simultaneously.

For $500, the compiler comes with libraries for each of the four memory models supported, a disassembler to examine code the compiler generates, header files, examples and sample programs, and complete documentation. Digital Research's linker is also included. Reader Service No. 22.
Lattice Inc.
2500 S. Highland Ave.
Lombard, IL 60148
312-916-1600

Oasys has recently announced C2PS, a program which automatically generates PostScript from C programs. The product is targeted at developers of applications for PostScript-based windowing systems like Sun Microsystems' NeWs and Adobe's Display PostScript; those who wish to use pre-existing C or C++ code in a PostScript environment; and anyone who uses PostScript but is more proficient in C.

Once C2PS has been used to generate a PostScript module, that code may be used directly (e.g., sent to a printer to produce a graphic image) or with other PostScript packages to build a complete application. C2PS sells for $2,995. Reader Service No. 23.
Oasys
230 Second Ave.
Waltham, MA 02154
617-890-7889

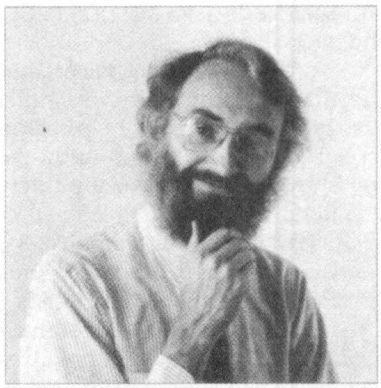
Software

Alde Publishing has introduced Ada-ROM, a CD-ROM based software library. The Ada software respository is one of several located on the Simtel20 Defense Data Network host computer at White Sands Missle Range in New Mexico.

Ada-ROM provides access to over 300 programs as well as data files, source code, templates, and information on the Ada programming language. It is accessed with an IBM PC XT, AT, PS2 or compatible and a CD-ROM player. Ada-ROM sells for $99 alone or $639 bundled with an Amdek CD-ROM player. Reader Service No. 28.
Alde Publishing
4830 W 77th St.
Minneapolis, MN 55435
612-835-5240

DDJ

SWAINE'S FLAMES



I was wondering if the first HyperExpo and StackMart, held in the home of the West Coast Computer Faire, would have a WCCF aura.

It was clear from the start that StackMart was smaller than the first WCCF: fewer than half the exhibitors, only a couple of dozen sessions. On the other hand, some of the same people (such as Larry Tesler) were there. As I wasn't at the first WCCF, I can't compare the moods of the two shows, but from what people tell me of that 1977 conference, I think this one had a little of the magic. And the exhibitors suggested that earlier market.

There were the books and magazines, demystifiers of the arcane. I picked up Keith Mathews and Jay Friedland's *Encyclopedia Mac ROM*, published simultaneously as stack and book by Brady/S&S (One Gulf+Western Plaza, New York, NY 10023), and Gary Bond's *XCMDs for HyperCard*, from MIS Press (1107 N.W. 14th Ave., Portland, OR 97209, 503-222-2399). I also got the Version 1.2 update to Jeff Stoddard's *HyperCard Scripting* from Walking Shadow Press (P.O. Box 2092, Saratoga, CA 95071, 408-354-7833); and the new issue of *HyperLink* magazine (P.O. Box 7723, Eugene, OR 97401, 800-544-0339) as well as the magazine-on-disk Stackazine (526 Jordan, P.O. Box 9, Bolingbrook, IL 60439).

There were the job opportunities. Contract developers were needed by Hall 4 (MCI mail 297-0147 or phone 408-973-7855), and Apple was looking for tech support people (Human Resources Dept. PM48H, 20525 Mariani Ave., Cupertino, CA 95014). There were stackware distribution opportunities with Heizer Software (P.O. Box 232019, Pleasant Hill, CA 94523, 415-943-7667 [office], 800-888-7667 [order line]). And MicroMaps Software was looking for authors to submit stack-based maps that it will distribute on a royalty basis with its open-ended HyperAtlas (P.O. Box 757, Lambertville, NJ 08530, 609-397-1611).

There were the services. The HyperMedia Group (1832 Woodhaven Wy., Oakland, CA 94611, 415-339-3322) offers consulting, application design, HyperTalk programming, and training. The Solutions Group (490 Santa Clara Ave., Redwood City, CA 94016, 415-368-1626) offers HyperCard-based presentations. And the Software Applications Group (P.O. Box 11089, Costa Mesa, CA 92627, 714-496-0881) offers HyperCard and CD-ROM development services.

There were the socially redeeming applications, such as the case study in the use of HyperCard to develop "prosthesisware" for an aphasic patient by Dr. Douglas Chute (Director of Neuropsychology, Drexel University, 215-895-1722).

There were the developers' tools. Symmetry (761 E. University Dr., Mesa, AZ 85203, 800-624-2485) makes HyperDA and the new HyperEngine that lets you open stacks from within your program. There was an XCMD and XFCN toolkit from Fidcor USA (728 Main St., Louisville, CO 80027, 800-247-7130). And there was HyperBase, an XFCN that adds relational database capabilities to HyperCard, from Answer Software (20045 Stevens Creek Blvd., Cupertino, CA 95014, 408-253-7515).

And there were user's groups, such as BMUG (1442 A Walnut St., #62, Berkeley, CA 94709), busily upgrading registered users to 1.2. BMUG has more than 50 disks of stacks today, attesting to the reality of stackware.

I wish I could list all the exhibitors here, but that would take about twice the space I have. There were some others that I must mention, such as DTP Advisor and The Electronic Whole Earth Catalog from Brøderbund (17 Paul Dr., San Rafael, CA 94903, 800-527-6263); PageMaker templates on CD-ROM that can be previewed from a HyperCard front end, from Image Express (2101 W. Chapman Ave., Orange CA 92668, 714-938-1070); and a Louvre catalog, 6,000 works of art, 35,000 images, and video commentary on selected works from The Voyager Company (2139 Manning Ave., Los Angeles, CA 90025, 213-474-0032).

And there was stackware there that probably shouldn't be mentioned, too. I had been worried about hyperglut, tons of bad stackware. Now as ever, George Morrow got it right: "Ninety percent of the software gets written in ten percent of the time. The next nine and a half percent takes ninety percent of the time. The last half percent never gets done. But the software still gets sold."

Yes, I think there was a lot of the early WCCF spirit at HyperExpo. I just don't know if that's good news or bad news.

Michael Swaine

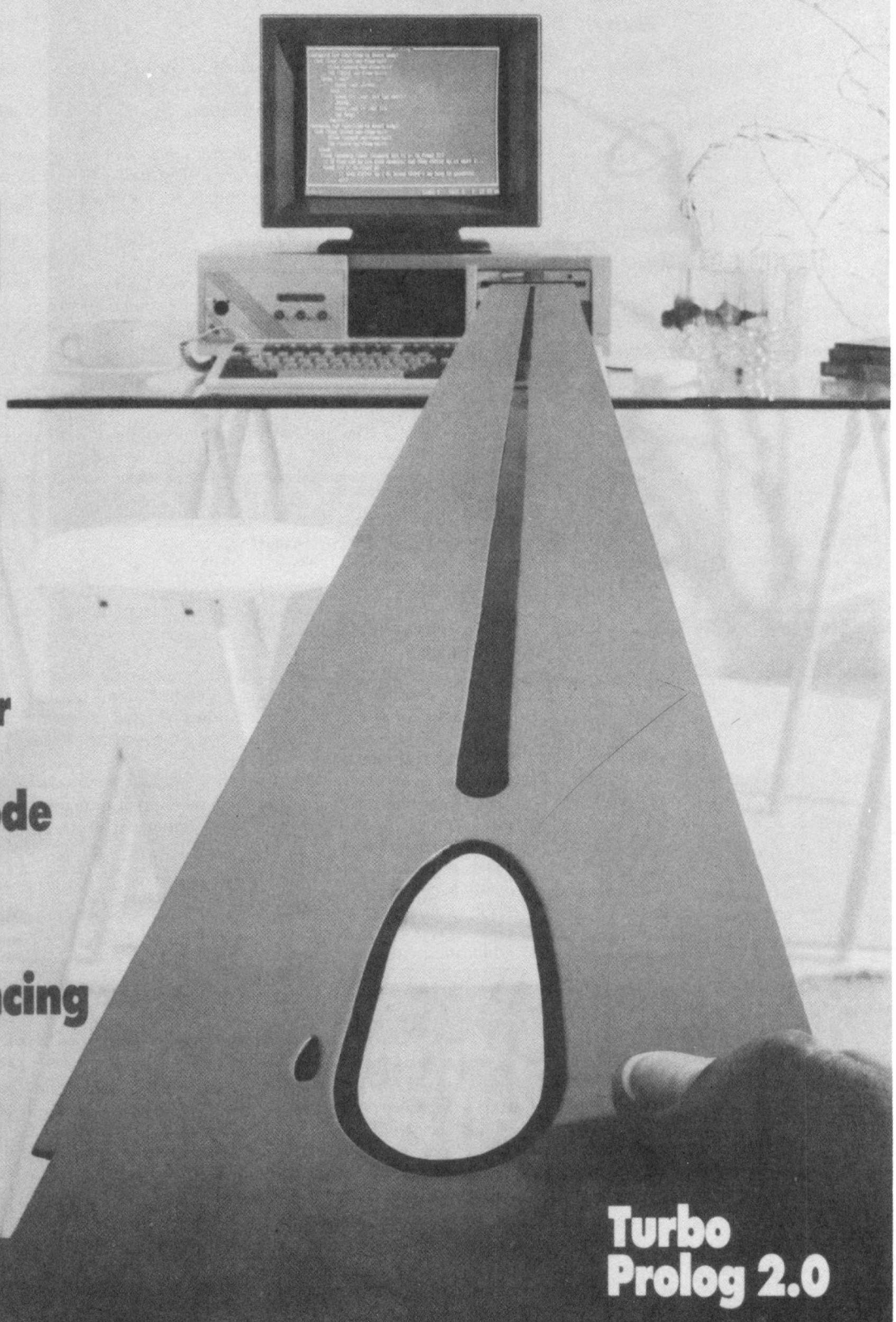
Michael Swaine
editor-at-large

#144 OCTOBER

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER



Languages You Can STRETCH

Extending LISP

80386

- An Assembler In FORTH
- Protected Mode

Symbol Cross-Referencing

Turbo Prolog 2.0

10
0 38351 16562 8

ARTICLES

Lisp ▶ **Adding Extensions to Lisp** 18
by Jonathan Amsterdam
Using Lisp's macro capability to add functions and control structures in your compiled code not only offers "grow your own" language extensibility, but lets you customize without a performance penalty.

Forth ▶ **An 80386 Assembler in Forth** 28
by John Dilworth
John develops an assembler written in Forth that takes advantage of the extended capabilities of the 80386.

80386 Protected Mode Initialization 36
by Neal Margulis
In addition to discussing how to get from real- to protected-mode and back, Neal shows how to define greater than 64K segments, and provides some general comments on the ins and outs of developing 32-bit software.

Assembly Language ▶ **A Double Cross for MASM** 46
by Steve Heller
Steve's multiple cross-reference solution will help you keep track of symbols and their uses in your assembly language programs.

REVIEWS

Prolog 2.0 ▶ **EXAMINING ROOM** 54
coordinated by Ron Copeland
Ernie Tello evaluates the most recent incarnation of Borland's entry into the world of AI languages — Turbo Prolog 2.0.

COLUMNS

Pascal and QuickBasic ▶ **STRUCTURED PROGRAMMING** 100
by Kent Porter
Earlier this year, Kent showed how to handle huge chunks of data. This month, he revisits the subject of large data arrays.

C PROGRAMMING 108
by Al Stevens
Al continues his multi-issue series on developing a communications program in C by detailing a window-menu facility and data entry screen driver. Al also ponders whether programming is an art and asks the question: "What is a C++ object anyway?"

THE FORTH COLUMN 116
by Martin Tracy
Martin resumes his wrap-up of some recent real-time programming conferences, including several important Forth-related announcements.

PROGRAMMING PARADIGMS 122
by Michael Swaine
Prior to this summer's IEEE Neural-Network conference, Michael was a bit confused about some of the aspects of neuro computing. Now, he's not sure whether he's confused or not.

FORUM

EDITORIAL 6
by Jonathan Erickson

RUNNING LIGHT 8
by Ron Copeland

LETTERS 12
by you

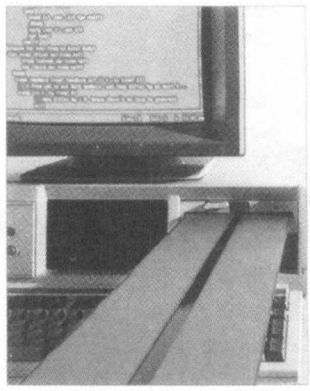
SWAINE'S FLAMES 152
by Jonathan Erickson

PROGRAMMER'S SERVICES

OF INTEREST 140
brief product descriptions

ADVERTISER INDEX 145
where to go for more information on products

PROGRAMMER'S MARKETPLACE 146
classified ads



Next Issue

"Looking good" is how to describe our November issue as we focus in on graphics. Some of the specific topics we'll be examining include shading subroutines, graphics standards, and image compression. We'll also take a look at ways to handle DOS and hear what our usual line up of columnists have to say about what's been on their mind, programming-wise.

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, except semimonthly in June and October by *M&T Publishing* Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3771, Escondido, CA 92025. **ISSN 0888-3076**

EDITORIAL

Even though 1989 is still a few months off, we've been planning the topics you'll be reading about in *DDJ* in the coming year and, based on what you've been telling us, we think you'll like some of the topics we'll be covering. As the following editorial calendar illustrates, 1989 will see us examining developments on topics we've covered before (real-time programming, for example) and taking an in-depth look at subjects, like windowing systems, that are becoming increasingly important to programmers.

January	Neural Networks
February	Real-time & Embedded Systems Programming
March	Windowing Systems
April	Memory Management
May	Structured Languages
June	Operating Systems
July	Graphics
August	Annual C Issue
September	*Modeling & Simulation*
October	Telecommunications
November	Parallel Processing
December	Object-Oriented Languages

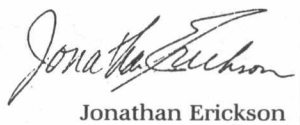
Although we've already started lining up articles on many of these topics (particularly those for the early part of the year), we'll still welcome ideas for specific articles you'd like to see. And if you'd like to write an article, so much the better. Just give me a call or drop me a letter describing what you have in mind.

What sort of articles are we looking for? We'll consider *something* on any of the above topics as well as articles about programs or utilities you've developed that will solve a particular programming problem. Stewart Nutter's "An Aid To Documenting C" in August, Ray Moon's September piece on "Arguments and Automatic Variables in Assembly Language," and Steve Heller's feature entitled "A Double Cross for MASM" in this issue, are all excellent examples of the kind of task-specific articles I have in mind.

But just because many of our features are DOS and C related, don't (falsely) assume that those are the only topics we want to cover; we'll give serious thought to proposals that relate to any topic that is important to programmers. (I'd like to see more articles dealing with Basic, for instance.) This includes articles covering non-DOS systems, especially the Macintosh. To digress on this point for a moment. . . .

Over the past few months, I've been compiling the "Archives" column (found on page 8 in this issue), something I've enjoyed doing for a couple of reasons. For one thing, reading through back issues of *DDJ* helps me keep in mind why this magazine was started in the first place. For another, it seems that, no matter which issue I pick up, I always learn something new. (Incidentally, if you have any favorite quotes from past issues, send them in and I'll include them in "Archives.")

In the process of compiling this month's Archives, I ran across a 1978 "Letter to the Editor" berating *DDJ* for spending too much time on 8080-based systems and not enough on 6502 and other such systems. It probably comes as no surprise to you that we hear the same complaint today — ten years later — the only difference being that the 8080-argument has evolved to the 80286/386 and the 6502-complaint is now the Macintosh environment. My response today is the same as then-editor Tom Williams: "Fact is, we'd be more than happy to publish stuff on 6502s and others if folks would write it up and send it to us. *Dr. Dobb's* exists to help pass software around to the eager hands of users."



Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Managing Editor *Monica E. Berg*
Senior Technical Editor *Kent Porter*
Senior Editor *Ron Copeland*
Editorial Assistant *Kathleen Evans Ralston*
Technical Editor *Richard Relph*
Contributing Editors *Al Stevens, Martin Tracy, David Betz*
Copy Editors *Kevin Shafer, Rhoda Simmons*
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Lisa Schneider*
Technical Illustrator *Lynn Sanford*
Typographers *Lorraine Buckland, Margaret Anderson*
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Asst. Circulation Manager *Andrea Weingart*
Newsstand Manager *Sarah Frisbie*
Fulfillment Coordinator *Francesca Martin*

Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura DiLazzaro*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Marketing Assistant *Sara Noah Ruddy*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Article Submissions: Send manuscripts and disk (with article and listings) to the associate editor.

***DDJ* on CompuServe:** Type GO DDJ

Subscription: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds with additional postage. Add $13 per year for surface mail to all countries or add $33 per year for airmail to Canada and Mexico; add $32 per year for airlift/surface to all other countries.

Customer Service: For subscription orders and changes of address call toll-free 800-354-8400 or write: *Dr. Dobb's Journal* subscription Dept., P.O. Box 3771, Escondido, CA 92025. For all other subscriber inquiries call 800-321-3333 (in California 800-331-4164, outside the U.S. 619-485-9623). For book/software orders call 800-533-4372 (in California 800-356-2002).

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright © 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

DDJ Offers Aid to Budding Mac Programmers

Now that summer newsstand shoppers, and the score of *DDJ*rs with bloodhound in their ancestry, have had a chance to sniff out *Dr. Dobb's Macintosh Special*, I thought perhaps it was time to give — in Michael's words —"credit where credit is due."

I'll begin with a nod and a wink to my ol' buddy Tyler followed quickly with a deep bow of appreciation to writers Dan Allen and Tony Meadow for the loan of their considerable talent and their avid support of Mac programmers. Thanks to their efforts, a number of aspiring Mac programmers were able to see some "in-depth" Mac coverage under the *DDJ* banner — even if it is a trifle more entrée than is customary for the good Doctor.

I also want to acknowledge the largely unsung efforts of Dave Lingwood, of the Apple Programmer's and Developer's Association, and Dick Hubert, of its umbrella company, the A.P.P.L.E. Co-op, without whom much of the occult knowledge of programming for the Mac environment would have never made it out of the hallowed halls of Cupertino. In particular, I want to personally thank Frank Catalano, APDA's public relations manager, who channeled an unstinting flow of information in my direction, so that I, in turn, could pass it along to you.

I'd be remiss if I failed to acknowledge the continuing commitment of the collective body of vendors who create and improve the rich selection of programming tools available to Macintosh programmers. Borland, Coral, Icom, Jasik Designs, SmethersBarnes, and Think (now a division of Symantec), to name a few, offer some of the best development software available on any platform. Hot new programming tools and revised versions of already excellent programs hit the streets almost weekly.

And don't forget Apple, whose commitment to increasing the performance of the hardware and to developing high-level object-oriented programming languages and other software tools should continue to make writing programs for the Mac easier and quicker (well, it should).

Response from readers of *Dr. Dobb's Macintosh Special* has been overwhelmingly positive. A couple of days ago we even received a call from the president of the London Mac Users' Group offering kudos and asking for more. In fact, the most common comment was actually more of a question: "What took *DDJ* so long?" I agree — *DDJ* should have done more to influence and support Mac programming. While we have devoted a smattering of coverage to the Mac since 1984 (Remember the "Fatten' Your Mac" article in January of '85 — we still get requests for it.), *DDJ* has probably been remiss in drawing attention to programming issues that relate specifically to the Mac. What do you think? Let us know.

In what, I'm sure, comes as no surprise to David Smith and his MacTutor writers, interest in Mac programming is alive and well and growing like an apple rolling down a snowy hill. Competent Mac programmers are still a rare commodity — especially ones who can stretch the envelope to provide applications which exploit the Mac interface in ways that extend computers into novel application arenas.

Ron Copeland
senior editor

ARCHIVES

Ten Years ago in *DDJ*

"The Forth system is like a root of a language which grows toward any application area for which it is used. . . . Forth is especially suited to real-time control applications, to large projects, and to small machines. Software development times are often a fraction of what would otherwise be required. Today, access to Forth can be hard to find. We expect increasing use of this language as it becomes more available." — *John S. James, "Forth Dump Programs,"* DDJ, *October 1978.*

Building Better Mousetraps

"An examination of technology advances indicates that hardware gains are due as much to the development of an appropriate tool technology as to the amount of direct resources expended. Efforts in the software domain, however, appear to have been directed much more toward producing immediately usable results than toward advancing the state of the art." — *Morris Dovey, "Introduction to PL/C: Programming Language for Compilers,"* DDJ, *March 1984.*

A Little Free Advice

"Software developers: these systems [CP/M-86 and MS-DOS] are similar enough that your products can run on both with only minor changes. Modularize your programs so as to encyst every system call in a subroutine of your own. Hackers and homebrewers: consider which system is the most up-front with information on its internals. Ordinary users: you really don't care, do you?" — *Dave Cortesi "CP/M-86 vs. MSDOS: A Technical Comparison,"* DDJ, *July 1982.*

Dr. Dobb's Journal of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Too Soft, Too Hard on MS Pascal 4.00

Dear *DDJ*,

I think you were both too hard and too soft in dealing with MS Pascal 4.00 in the June issue.

Kent Porter was too hard on Microsoft when dealing with comparisons to Borland's "Turbo" Pascal 4.00. Microsoft's implementation is, in my opinion, much closer to the ISO standard, and because it uses an external linker and a recognized object module format it is much more universal. I can, and have linked modules written in MS Pascal with routines written in assembly language and MS C — a task much more difficult, if not impossible, when using "Turbo" Pascal. I also agree with Charles Linett's comments about the defects in the Borland manual. Nevertheless, I do use the implementation on account of its graphics library.

Porter was too easy on Microsoft because he did not compare this release with the previous version — 3.32. In this regard, I can find no justification for the change in major version number. As a devoted MS fan, I upgraded at a cost of $100 (Canadian). I expected to find, in addition to the support for Windows and OS/2, some real improvements in the implementation of the kind that were evident between Fortran 3.32 and 4.00. Shouldn't we have received 1. *new* manuals, 2. implementation of the "Unimplemented Features" (how can a feature be unimplemented?), and 3. a PL driver equal in features to the Fortran and C drivers — both in the use of environment variables and options? Why was there no graphics library similar to the one available with C Version 5.0x?

Charles Chapman
London, Ont., Canada

A Surgeon Wouldn't Use a Swiss Army Knife

Dear *DDJ*,

I would like to clarify Bruce Tonkin's dismissal of the Forth language as being "generally hard (or downright impossible) to read without significant study" (July issue). He points out that "spaghetti code is always the programmer's fault." The same argument could be applied to readable code. If a portion of Forth code is unreadable, it is probably due to the outrageous flexibility offered by the language: whereas most languages dictate a fixed syntax, Forth does not.

While most languages allow only certain characters in the names of user-defined procedures, Forth carries no such restrictions. Since a Forth program is built by extending the language, these factors have a huge impact. A good Forth programmer can produce very readable code, and a bad Forth programmer can hack out chaos. That's the price of flexibility.

As to Basic's being superior to other languages on the basis of having "so many things built-in," nonsense. With the abundance of C libraries on the market, a C programmer can have virtually any function, user interface, btree manager, etc. without carrying along a bulk of dead code.

"If C is a scalpel, Basic is a Swiss Army knife." I couldn't agree more. But a surgeon doesn't use a Swiss Army knife.

Dave Ruske
Milwaukee, Wisc.

Dear *DDJ*,

Bruce Tonkin raises some interesting points on different Basic compilers in his July article "Getting Down to Basics," but his annoyingly parochial attitude about the 8087 chip makes his article useless for evaluating the different Basic compilers for engineering applications. Where does he get his information that "most" computers do not have it? It is essential for any PC used for engineering programs and is one of the first additions I made to my computer. Most, if not all, of the computers available to students in the Colorado State University College of Engineering have hardware and math chips. It certainly speeds up Lotus 1-2-3 calculations.

In his table and text on page 60, Tonkin claims that Turbo Basic is 160 times slower than Microsoft Basic in long integer additions, without specifying whether or not he used the 8087 chip in his benchmarks. Since it is critical to know whether a mathematical benchmark operation was or was not performed with the 8087 chip, it is unprofessional to assume that everyone else has only bought the same enhancements as Tonkin and his friends.

I hope that this does not represent the caliber of all articles in *DDJ*.

Don Baker
Ft. Collins, Col.

Debatable Debugging

Dear *DDJ*,

The only debugger I know of that is set up to handle device drivers as Gary Hornbuckle needs to do (July issue) is the Quaid Analyzer which is advertised on page 85 of the same issue in which his letter appears. I've had fairly good results with it in the year I've used it; the manual is a tad terse, but those who know enough about systems software to write loaders and drivers should be able to dope it out. (Its price has doubled in the past year—looks as though the wayward jet stream has blown some of that good old U.S. greed north of the border.)

As for the detailed structure and loading process for .EXE programs, I believe that is fairly closely-held proprietary information; Dr. Hornbuckle should not be surprised to receive a warning shot from the litigious folk at Microsoft. However, the Quaid Analyzer ought to make it fairly simple to step through the loading process and observe what is done to the program and how control is transferred to it.

I assume by "DOS function 75" he

means function 4Bh; it seems to have been Big Chief Face-on-the-cover who started this silly—and potentially misleading—custom of referring to DOS functions be decimal instead of hexadecimal numbers. Why does *DDJ* go along with it?

John S. Cikoski, Phd
Columbus, Ohio

Dear *DDJ*,

I have a few suggestions for Dr. Gary Hornbuckle (*DDJ* "Letters," July issue) on information about the format of EXE files and debugging device drivers.

First, the best book about programming with DOS is *Advanced MS-DOS* by Ray Duncan (Microsoft Press, 1986). It has a detailed description of the EXE file header, though not of the relocation tables and such. It probably does not have enough information for you to write your own loader, but it may have enough for you to use DOS function 75 for your purposes. It also has a good description of the structure as a device driver.

Second, for debugging a device drive, I highly recommend the Periscope debugger from the Periscope Company. It is a memory-resident debugger that can be loaded as a device driver for debugging other device drivers or used to debug normal programs. I have not actually used it to debug a device driver, but I have used it extensively for debugging non-DOS real-time multitasking systems as well as ordinary DOS programs. Periscope is available as a software-only debugger or with protected-memory hardware to keep the debugger from being overwritten, or with hardware-level breakpoints and an instruction trace buffer. (I've used all three configurations.)

It has many other capabilities and is generally a pleasure to use.

Jim Hewitt
Beltsville, Maryland

"Binary Bill" Booker lends new definition to the term personal computing

Adding EXTENSIONS to Lisp

Lisp, more than any other programming language, gives you the power to extend the system

by Jonathan Amsterdam

More than any other language, Lisp gives you the power to alter and augment the system — and, if you are not careful, the rope to hang yourself in the process. There are many reasons for Lisp's extensibility: interpreted execution, which permits you to use the full power of Lisp when writing macros; a uniform representation of programs and data, which allows programs to be manipulated easily; and a simple and uniform syntax, which allows macros to take on the appearance of built-in functions and control structures. The extensible nature of Lisp is one reason, in fact, that MIT professor Joel Moses once compared it to a ball of mud: no matter how much mud you add to Lisp, he said, it still looks like a ball of mud. In this article, I examine the ways you can extend the language using macros.

Characteristics of Lisp Macros

With two exceptions, a Lisp macro is evaluated just like any other Lisp function. The first exception is that the macro's arguments are not evaluated, the second exception is that the value of the result is used in subsequent

Jonathan Amsterdam is a graduate student at MIT's artificial intelligence laboratory. He has articles published in several magazines, including the April, 1988 issue of DDJ. *He can be reached at Room 814, 545 Technology Squ., Cambridge, MA 02139.*

(continued from page 18)

computation. The following macro, which is a useful device for those programmers who find Lisp's function names too cryptic, illustrates these exceptions:

```
(defmacro first (x)
   (list 'car x) )
```

This macro makes first a synonym for car, so *(first '(a b))* causes *first* to execute on the list *'(a b)*. The operation first returns *(car '(a b))*, which is then evaluated to yield *a*. Used interpretively, macros are no cheaper than functions, but when Lisp code is compiled, the initial evaluation (called *macro expansion*) is done at compile-time, so there is no loss of efficiency. For all practical purposes, using *first* is the same as using *car*. Therefore, you can think of Lisp macros as extensions to the compiler.

Note that you could have used the backquote notation to create the macro just given like this:

```
(defmacro first (x)
   `(car ,x) )
```

The first symbol on the second line is not the familiar Lisp quotation mark, but is a backquote (the accent grave to francophones). Backquotes act just like quotation marks, except that any form inside a backquote that's preceded by a comma is evaluated.

You can see how Lisp macros can be used to replace functions, or sequences of statements, by a single call. These sorts of macros are commonly used, but they're not terribly interesting; you can get the same effects with C's macro preprocessor. Time is too precious to waste discussing them, so let's move on to more exciting things. Many of the macros I'll be discussing already exist in full-featured Lisps, like Common Lisp, but for expository purposes I'll assume a very bare-bones Lisp, with no control structures except for function calling, *cond*, *prog*, and *go*. So to write a simple loop, you'd have to write something like the code in Example 1, this page. There's nothing wrong with this code, except that it's ugly. If you had a *while* loop, you could have instead written the much prettier code in Example 2, this page. To define a *while* loop, you can write a macro like the one in Example 3, this page. There is, however, some syntax in Example 3 that I need to explain.

```
(prog (i)
      (setq i 1)
   loop
      (cond ((>= i 10) (go end)))
      (print i)
      (setq i (+ i 1))
      (go loop)
   end)
```

Example 1: *A simple, but messy loop*

```
(setq i 1)
(while (<= i 10)
  (print i)
  (setq i (+ i 1)))
```

Example 2: *A cleaner approach than Example 1, using a while loop — if you have one*

```
(defmacro while (test &rest body)
  `(prog ()
      loop
         (cond ((not ,test) (go end)))
         ,@body
         (go loop)
      end))
```

Example 3: *Defining a while macro*

```
(prog ()
   loop
      (cond ((not (>= i 10)) (go end)))
      (print i)
      (setq i (+ i 1))
      (go loop)
   end)
```

Example 4: *A more efficient way of constructing the code in Example 1*

```
(do ((i 1 (+ i 1)))
    ((>= i 10))
  (print i))
```

Example 5: *A for loop*

The *while* loop looks just like ordinary, built-in Lisp. Since all of Lisp — even control flow — looks like functions applied to arguments, macros fit in quite nicely. In languages like C that have a variety of syntactic constructs, macros, which usually have the syntax of function calls, look out of place when you use them to define control structure.

You can improve the *while* loop by writing a *for* loop, thereby allowing the computation to be expressed as shown in Example 5, this page. The complete macro is shown in Listing One, starting on page 56. In the Listing, I've assumed that *first*, *second*, and *third* are defined either as macros or functions in the obvious way. This macro is the first one illustrated so far that performs some computation instead of immediately returning a form. In this case, the computation is trivial — just taking apart its first argument — but in general, you can perform any computation.

To illustrate further, examine the macro in Listing Two, page 56, that performs simple loop unrolling: if the macro notices that the bounds of the iteration are within one of each other, the macro won't generate loop code at all. This macro is quite interesting: it will generate different code at compile-time depending on its arguments. If Lisp macros are compiler extensions, then we have just written part of an optimizing compiler.

(continued from page 20)

The Macro setf

We'll return to iteration in a moment, but first I'd like to discuss *setf*, a clever macro that's been lurking around for years and that has finally stepped into the limelight because of its incorporation into Common Lisp. *setf* is interesting because it's an extensible macro. In its simplest form, you can use *setf* like *setq*:

(setf a 3)

But you can also use *setf* to set almost anything, such as elements of lists or arrays:

(setf (car a-list) 'x)
(setf (aref an-array 3) nil)

You can extend *setf* so that it can handle new forms. The key to providing this extensibility is Lisp's property-list feature. The *setf* macro does no work by itself: instead, it looks at the first element of the form that it receives, and calls the function on the *setf-method* property of the symbol. Before making the call, however, *setf* expands any macros in the form by calling the built-in function *macroexpand* shown in Example 6 on this page. The *setf methods* perform all the work. Example 7, page 24, shows how the macro for *car* might look. This macro expands *(setf (car a-list) 'x)* into *(rplaca a-list 'x)*. Before using it, we must install it on *car*'s property list. In most Lisps, you'd use *putprop* for this, but just to show you how far Common Lisp has incorporated *setf*, the only (documented) way to add a property to a property list in Common Lisp is as follows:

(setf (get 'car 'setf-method) 'car-setf-method)

Example 8, page 24, shows how to define the *setf-method* for *aref*, assuming the function *aset* exists.

```
(defmacro setf (form value)
  (setq form (macroexpand form))
  (cond
    ((symbolp form)
     `(setq ,form ,value))
    (t
     (funcall (get (car form) 'setf-method)
              form value))))
```

Example 6: *The setf macro*

An Extensible Iteration Macro

You can combine iteration with extensibility to produce an extensible iteration macro. I'll call this macro *for* as well, because it will subsume our earlier attempt. This macro will let you say something like:

(for (i from 1 to 10) . . .)

to iterate over a range of numbers,

(for (el in '(a b c)) . . .)

to iterate over the elements of a list,

(for (subl on '(a b c)) . . .)

EXTENSIONS TO LISP

to iterate over successive sublists — in this case the lists

(a b c), (b c), (c)

and

( )

and

(for (ael array-elements of a) . . .)

to iterate over the elements of an array.

The syntax of our macro is fairly straightforward: after *for*, there is a list consisting of a variable name, a keyword that indicates the type of iteration, and some additional items describing the iteration. You could write the macro to test for each keyword in turn and output the appropriate code, but that wouldn't allow the user to extend it.

```
(defun car-setf-method (form value)
  `(rplaca ,(second form) ,value))
```

Example 7: *Using setf-methods for car*

```
(defun aref-setf-method (form value)
  (let ((array (second form))
        (indices (cddr form)))
    `(aset ,array ,value ,@indices)))
```

Example 8: *The setf-method for aref*

```
(defun num-expander (var from-to)
  (let ((from (first from-to))
        (to (third from-to)))
    (list `((setq ,var ,from)) ; Initialization
          `(if to
               `(> ,var ,to)
               nil)
          `((setq ,var (1+ ,var))) ; Stepping
          nil)))
```

Example 9: *The function for iterating over numbers*

```
(defun list-el-expander (var list)
  (setq list (car list))
  (let ((sublis-var (gensym)))
    (list `((setq ,sublis-var ,list))
          `(setq ,var (car ,sublis-var))
          `(null ,sublis-var)
          nil
          `((setq ,sublis-var (cdr ,sublis-var))
            (setq ,var (car ,sublis-var))))))
```

Example 10: *The function for iterating list elements*

```
(for (i from 0 to (- (length list) 1)
      (setf (aref array i) (nth i list))
```

Example 11: *One way to iterate over a list*

```
(for (el in list)
     (i from 0)
  do (setf (aref array i) el))
```

Example 12: *A better way to iterate over a list*

Instead, we'll use the property-list feature. This new macro (implemented in Listing Three, page 56) looks on the property list of the keyword for the value of the property *for-expander*; this should be a function of two arguments, the variable name and the item following the keyword. The function will return a list of three items: initialization code that will be executed before the loop begins; a piece of test code that will be run before each iteration; and update code that will be run after the body executes on each iteration. The initialization code and the update code should be lists of statements. Let's do two of the earlier mentioned four examples. The function for numbers, is shown in Example 9, this page.

There is one additional feature: you can get an infinite loop by omitting *to* and writing only *(i from 3)*. Doing this can be useful, as you'll see later. Next, we need to install *num-expander* on the property list of *from*:

```
(setf (get 'from 'for-expander) 'num-expander)
```

The function for list elements is given in Example 10, this page. This function is a little tricky because the iteration variable, which steps over sublists, is hidden. Since the macro needs to come up with its own variable, it creates a brand new symbol using the Lisp function *gensym*. The other two examples of *for*, along with any others you can think of, are left as exercises.

Here's another small extension to the *for* macro. Often, you might like to iterate over two different sequences in the same loop. For instance, imagine that you wanted to put the elements of a list into an array. One approach is shown in Example 11, this page, but this technique will perform extra work, using *cdr* down on the list from the beginning for each element. A better approach is to iterate over the list and the integers simultaneously, as shown in Example 12, this page. Since *for* is defined to terminate as soon as any of the iteration clauses terminates, you can omit the *to* in the counting clause. Note the word *do*: You must include it to distinguish the body of the *loop* from the clauses. Alternatively, you could have enclosed all the clauses in a list, but sometimes it is worth adding a keyword to get rid of a layer of parentheses.

The *for* macro is actually a small example of what you can do with iteration. A bigger step is the *loop* macro of Symbolics Common Lisp, whose fine-grained model of iteration allows for more complex control-flow patterns. For example, the functions you write to extend *loop* must return not 3, but 6 values. The complete code for *for* can be found in Listing Four, page 58.

Records in Lisp

Let's turn from control abstraction to data abstraction, and implement a simple record package in Lisp. Assume your Lisp has arrays and you want to implement records in terms of those arrays. You would probably want to write:

```
(defrecord complex
   real imag)
```

to define a record for complex numbers,

```
(make-complex 3 4)
```

to construct the number with a real part 3 and an imaginary part 4,

```
(complex-real c)
```

to extract the real part of the complex number *c*, and

```
(setf (complex-imag c) 5)
```

to set the imaginary part of c. Furthermore, imagine that you'd like accessing components to be no more expensive than array references, so the accessing functions need to be macros. What's interesting about *defrecord*, then, is that it's a macro-generating macro. It is actually very straightforward to write (see Listing Five on page 58). The only tricky part — and believe me this comes only with considerable macro-writing experience — is keeping straight how many times to evaluate something.

Since *defrecord* returns more than one form, it wraps them all in a *progn*. First it defines the *make* function and then, for each record component, an accessor macro. Because *setf* already knows about *aref* and expands macros automatically, you don't have to do anything else to get *setf* to work correctly on the records.

There are scores of extensions you could add to *defrecord*. I'll describe the way to implement a few of them, leaving the code to you.

As it stands now, records are just arrays and accessor functions are just array references, so there's no protection: you could use *complex-imag* to access a record of another type. To ensure that you are using record selectors only on records of the appropriate type, you could put the name of the record into the zeroth slot of each record created, and have the accessor functions check this tag before they do anything. Since each accessor macro will no longer be a simple *aref*, you'll have to write *setf* functions as well, which will also need to check the record type.

At the same time, you could also address another typing issue. Right now, you could put a Lisp symbol into a slot of a complex record, where only numbers should go. This ability, of course, is a problem (or feature, depending on your point of view) with Lisp, but you can have it both ways. Let's extend the *defrecord* syntax so that

```
(defrecord complex
   (real number)
   (imag number) )
```

is allowed, so that you can place only numbers in the record's slots. Now *setf* functions, in addition to checking that they're being used on the right kind of record, also need to check the type of value they're supposed to store into a slot. To associate code — in this case, the *numberp* function — with the names of types, use property lists; that way, your type system will be extensible.

All this type-checking will take time, so you might want to provide an option to turn it off. You could easily arrange matters so that

```
(defrecord (complex :no-type-checking)
   (real number)
   (imag number) )
```

would result in no generation of type-checking code. For compatibility with other records' type-checking code, you'd still want to put the record name in each record you create.

Another problem is that the records print like arrays. If you tag the records with their names as I suggested earlier, you can take care of this problem by replacing your Lisp's primitive printing function with one of your own. On receipt of an array, your own function first checks to see if there's a symbol in the zeroth element, and then looks on that symbol's property list for a printing function. While adding this function, you can provide a default printing function for users of *defrecord*, so they don't have to do any extra work to have their records print nicely. Before long, you'll wonder how you ever lived without macros.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 56).

An 80386 Assembler In Forth

The portability of Forth means this assembler can be used with a variety of processors, not just the 80386

John B. Dilworth

The 80386 assembler, written in Forth (Forth-83 Standard), presented in this article will benefit not just Forth programmers, but anyone interested in learning about or using the rich, efficient instruction set and highly versatile address and operand modes of the 80386 microprocessor. The 80386 can be programmed using either a "flat" model of memory organization, in which up to 4 gigabytes of memory are physically addressable as a single array, or the segmented model familiar to 8086 and 80286 programmers (whose code runs unchanged on the 80386). If you then consider the CPU's sophisticated on-chip memory-management, protection, and multi-tasking facilities, and its strategic role in the MS-DOS OS/2 world, you will have a microprocessor that all serious programmers must understand in depth. This assembler is both a helpful learning aid and a useful tool that is easy to modify or enhance.

John B. Dilworth is a professor in the Department of Philosophy, Western Michigan University, Kalamazoo, MI 49008, and specializes in Artificial Intelligence research.

The assembler implements the full 80386 instruction set, including the privileged instructions available only in protected mode. Thus, it is a complete 80286 and 8086 assembler, since the 80286 instruction set is a proper subset of that of the 80386. As written, you can immediately use this assembler in real, non-protected mode with the popular F83 public-domain Forth system of Laxen and Perry. You can easily adapt it to other Forth (or non-Forth) environments, or you could modify it to produce stand-alone object-code files with little difficulty. Because the assembler is written in high-level Forth (no CODE words), the code itself is portable and non-processor-specific; because the F83 system has been implemented on 8080, 8086, and 68000 systems, the assembler is also a cross-assembler when run on such non-80386 processors.

Forth Assemblers

As with other Forth assemblers, this assembler would normally be used as a supplement to the resident compiler in a Forth system, whenever speed or other efficiency considerations point to native machine code as the best solution. The resident compiler itself produces threaded Forth code from Forth source which is organized as Forth words in the standard Forth dictionary structure. The assembler produces machine code for the host microprocessor, but also compiles it into the dictionary as the parameter field of a Forth *CODE* word. This word has a standard Forth header with name, link and code fields. You can then invoke machine-code definitions by name in Forth source code in the same way as ordinary Forth words. *CODE* words are written in the form

```
CODE NAME operands + opcode
                        . . . END-CODE
```

with the *CODE* automatically invoking the assembler at compile time and *END-CODE* terminating it.

You can find this integrated approach to combining high-level and machine-level code constructs in the structure of Forth assembly language itself. In addition to Forth assembly equivalents of the directives, opcodes, and operands of a conventional assembler, the full power of the Forth interpreter is always available for con-

stant and variable definitions, address calculations, procedure definition and invocation, or any special tricks you might want to do while assembling code. Forth equivalents of such high-level constructs as *WHILE-DO*, *REPEAT-UNTIL*, and *IF-THEN-ELSE* are also available (see screen 59, in Listing One, page 60). You may mix these freely with more conventional assembly language forms. One desirable result of this approach is that the labels conventional assemblers use for conditional or unconditional jumps become unnecessary, just as you can eliminate *GOTO*s in high-level languages by structured-programming techniques.

To make this 80386 assembler as accessible as possible, I have adopted many of Michael Perry's sound, efficient methods, and basic Forth-assembler vocabulary as found in his original 8086 assembler (supplied with the F83 system). However, since the changes involved in moving up from the 8086 to the 80386 are so extensive, I've had to rework virtually every Forth word and introduce many new ones. (The new assembler is over four times the size of the original.) With regret, I have resisted any temptation to optimize for size or speed in favor of a plain, more comprehensible style.

Fortunately the F83 Forth system is well-documented not only by the authors, but also in painstaking detail by C.H. Ting in his book *Inside F83* (San Mateo, CA: Offete Enterprises, 1986). Thus, if you need more basic details concerning Forth and Forth assembly language than this article provides, you have plenty of good resources available including the selections *A Forth Assembler for the 6502*, by William F. Ragsdale (pages 203 – 214) and *A 68000 Forth Assembler*, by Michael A. Perry (pages 193 – 202), both in *Dr. Dobb's Toolbox of Forth* (Redwood City, CA: M&T Books, 1986).

In this article, I shall concentrate upon the specifics of the 80386 that go beyond those of the 8086, along with the additional Forth structures that you need to implement them. For more general information on software aspects of the 80386 and its instruction set, see *Programming the 80386* by John H. Crawford and Patrick P. Gelsinger, (San Francisco, CA: Sybex Inc., 1987) and *80386 Programmer's Reference Manual* (Santa Clara, CA: Intel Corp., 1986).

80386 Instruction Assembly

An assembler's job is to translate instruction mnemonics and associated operands into machine code for a specific processor. As shown in Figure 1, page 29, for the 8086, the target or output bytes in the translation comprise — in order of run-time processing and from low to high memory — a possible segment-override prefix byte, 1 – 2 opcode bytes, 0 – 1 *ModR/M* bytes (described later), 0 – 2 displacement bytes, and 0 – 2 immediate-data bytes. (All of these bytes could additionally be prefixed by one of several repeat prefixes or the *LOCK* prefix to exclude bus interference by other processors.) Note also that 80x86-series processors store and process either multiple-byte addresses or data with low-order bytes before (at lower addresses than) higher-order, more significant bytes.

You can immediately use this assembler in real, non-protected mode and easily adapt it to other Forth (or non-Forth) environments

The 80x86 *ModR/M* byte has three fields as shown in Figure 2, on page 29. Most-significant bits 6 and 7 make up a MOD field, which combines with the 3-bit R/M field (least-significant bits) to specify 32 possible values to indicate 1 of 8 registers or 1 of 24 indexing modes. The REG field uses the next 3 bits following the MOD field, and specifies either a register or further opcode information. Its meaning is determined by the opcode byte(s) of the whole instruction. Finally, as already indicated, the 3 least-significant bits make up an R/M field whose interpretation depends on the value in the initial 2-bit MOD field.

As Figure 3 (below)shows the kinds of output bytes that an 80386 assembler produces include all of the 8086 kinds, along with several new or 32-bit-extended varieties. Two new prefixes — the Address-Size and Operand-Size — are available, which you can use to override the default *Use type* of a segment (indicating whether it is used for 16-bit or 32-bit code or data) for the specific instruc-

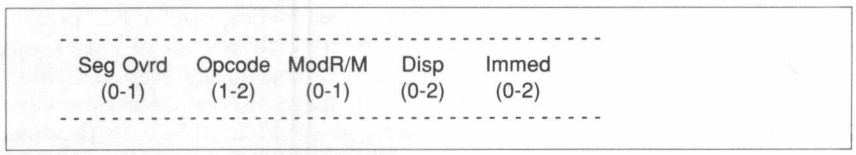


Figure 1: *8086/286 instruction encoding*

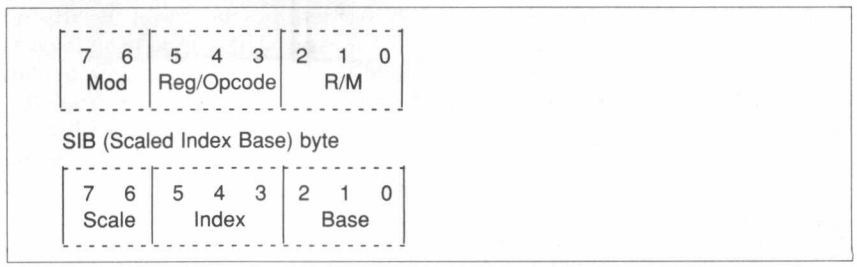


Figure 2: *ModR/M and SIB byte*

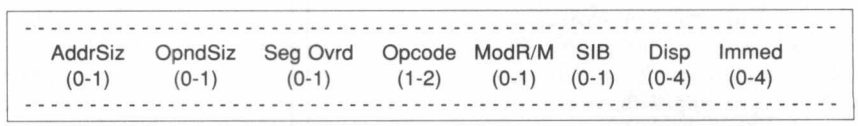


Figure 3: *80386 instruction encoding*

(continued from page 29)

Mod	Base	Operand	Forth Example
00	not 101	[base+(scale*index)]	[EAX+ECX] or [EAX+ECX] *1
00	101	[disp32+(scale*index)]	234,567 [EAX] *2
01	any	[base+(scale*index)+disp8]	7 [ESP+ECX] or 7 [ESP+ECX] *1
10	any	[base+(scale*index)+disp32]	89,567 [EDX+ECX] *4

Note: As usual, default segment register is DS: (data segment) for all registers but ESP and EBP, whose default is SS: (stack segment.)

Figure 4: *Cases needing SIB byte (R/M = 100)*

tion following the prefix. There are also two extra segment-override prefixes that correspond to the new general-purpose segment registers FS and GS available with the 80386. The remaining new item consists of a possible *SIB* byte (Scaled Index Base byte, described later) occurring immediately after the *ModR/M* byte; if present, the *SIB* byte is signalled by a binary 100 in the R/M field of the *ModR/M* byte. You can find these 32-bit extensions in the displacement bytes (now 0 – 4 bytes) and the immediate-data bytes (also 0 – 4 bytes.)

The 80386 *SIB* byte is needed in those cases when memory operands are too complicated to be represented by a single *ModR/M* byte as shown in Figure 2. In these cases, a Based Indexed or Scaled Indexed indirect memory mode is required; which are the [reg] *x,[reg + reg] and [reg + reg] *x cases in the assembler, as shown in Figure 4, page 30. The relevant information is encoded in three fields of the *SIB* byte. First, bits 6 and 7 make up the SS field, specifying the *scale factor*, which is a multiplicative factor that facilitates efficient addressing of offsets for different data sizes. The middle 3-bit field specifies the index register, while the lowest 3 bits give the base register.

Forth Structure and Code Walk-Through

Given the required output or object-code byte and bit-field structures specified earlier for an 80386 assembler, let's look at the way this Forth assembler goes about translating input instructions and operands into the required forms. First study Figure 5, page 33, which compares Forth assembler syntax (the 80386-specific parts are my own invention) with the more conventional syntax of non-Forth assemblers. Note the mirror-image quality of the Forth relative to the conventional syntax: this quality relates to Forth's stack-based postfix mathematical notation (*3 + (4 * 5)* is *3 4 5 * +* in Forth). Forth assemblers avoid a separate parsing stage for assembly source code by defining all of the operands and opcodes as full-fledged Forth words. These words can then be processed by the regular Forth interpreter. (However, you easily can add a separate parse routine as a preprocessor to this assembler if you prefer the conventional syntax.

Operands leave values on the data stack, which are then processed by a Forth opcode word — hence, the need to have the operands before the opcode in the Forth syntax. Source operands precede destination operands, and numeric arguments precede operators within each operand. For example, Figure 5 includes the following case:

6 [EBX] EDX BSF BSF EDX,[EBX] + 6

BSF is a new 80386 instruction that scans forward in the source operand (from bit 0 upwards) to find the first set bit. If a set bit is found, the destination operand is loaded with its bit index (the zero flag is also set). In this case, the source is an indirect memory operand that specifies an address (in the default segment DS) at 6 + the contents of *EBX*; the destination operand is register *EDX*.

The Forth interpreter handles this case as follows. The literal constant *6* goes on the Forth stack. Next, the Forth word *[EBX]* (defined on screen 3) is executed and places the special constant 3611d or E1Bh on the stack, the upper 8 bits of which (00001110b) identify it as a register-indirect constant, and the lower 8 bits (00¦011¦011b) of which both identify the register and are used via masking to set appropriate values in the corresponding *ModR/M* or *SIB* byte fields. (Internally, most assembler calculations are carried out in octal because base 8 is best for representing 3-bit fields.) Next *EDX* (defined on screen 2), which places its own special constant 2578d = A12h (the upper 8 bits 00001010b defining it as a 32-bit register, the lower 8 ¦ 00¦010¦010b again defining a masking template) on the stack. As shown in screen 43, everything is now set up for the execution of the Forth opcode word *BSF* as defined by 19MI:

```
19MI : CREATE C, DOES> C@
            ONPREFX 17 ( octal) C, C,
   OVER REG? ( source a reg also?)
      IF RR, ELSE ( mem source)
                  MEM, THEN WRAP ;
HEX ( change base) BC 19MI BSF
```

19MI is a *defining word* which cre-

	Forth	Conventional (MASM, etc.)		
	ECX	DIV	DIV	ECX
CX AX	MOV	MOV	AX,CX	
ECX EAX	MOV	MOV	EAX,ECX	
37 # BX	ADD	ADD	BX,37	
123,456 D# EBX	SUB	SUB	EBX,123456	
ADRS32 D#) ECX	BSR	BSR	ECX,ADRS32	
ES: DX 0 [BX]	CMP	CMP	ES:[BX],DX	
6 [EBX] EDX	BSF	BSF	EDX,[EBX] + 6	
345,678 [EDX+EAX] *4	PUSH	PUSH	[EDX+EAX*4] + 345678	

Figure 5: *Forth versus conventional assembler syntax*

ates the word *BSF*, stores in it byte 2 (BCh) of its opcode, and sets it up to use the code after the word *DOES>* in *19MI* itself. This economical and powerful Forth technique is used widely in the assembler, wherever a *group of* similar words needs to be defined. *BSF* works as follows: *C@* fetches its stored opcode; *ONPREFX* stores the opcode in the variable *OP*; sets a flag (OPSET ON) for a 2-operand opcode; and calls *PREFX* (screen 22), which invokes *ADR-PREFX*, *OPNDPREFX*, and *SEGOVR?* that check the stack contents to see if address, operand, or segment-override prefixes are needed. If they are, *BSF* then assembles them if necessary. *ONPREFX* concludes by fetching the stored opcode, which *BSF* (17 C, C,) assembles after the first opcode byte (octal 17 = Fh.) Note that throughout this and later processing, the algorithms used must ensure that the assembler does not confuse the special constants on the stack with memory displacements or the immediate data also found there. The postfix structure of the stack contents make this straightforward: items nearer the top determine the interpretation of those further down.

Having dealt with prefixes and opcodes, *BSF* now checks (OVER REG?) whether the next-to-top source stack item is a register constant; in this case, it is not, so *ELSE MEM* applies. *MEM*, (screen 16) organizes the fairly complex series of tests and decisions (screens 11 – 16) required to sort out and assemble. In the present case, MEM32 (screen 15), which handles the *disp [REG32]* cases, is called, *RMID OVER RLOW OR* sorts out the required fields of the destination and source operands (leaving 00¦010¦011b); *DOUBLE?* tests for a 32-bit displacement; and, after more tests, assembly is completed with the mode-1, byte-displacement code *100 OP, C*, which *OR*'s octal 100 (01¦000¦000b) with the previously formed *ModR/M* fields to give the complete *ModR/M* byte 01¦010¦011b = 53h, followed by the 1-byte displacement 6 assembled by *C*. The complete code assembled is *hex 67 66 0F BC 53 06*, here *67* and *66* are the address and operand-size prefixes, which are needed because of the 32-bit operations in a 16-bit data segment. *0F BC* are the required opcode bytes.

Because the assembler is written in high-level Forth (no CODE words), the code itself is portable and non-processor-specific; because the F83 system has been implemented on 8080, 8086, and 68000 systems, the assembler is also a cross-assembler when run on such non-80386 processors.

Features and Immediate Uses for the 80386

What can you do with the 80386 running in real mode (that is, its default power-up mode) under DOS? Clearly the full power and flexibility of its 32-bit processing requires an operating system designed to support all its features; but in the meantime there are useful enhancements available. For example, one of the pains of working with the segmented 80x86 architecture is not only the 64K segment limitation, but only having one extra segment register (ES) to work with. The new FS and GS segment registers should improve the efficiency of data transfer involving more than two areas.

With the 32-bit registers, another improvement is that you can use any one as a base or index register in indirect memory operations (previously, you could use only *BX, BP, SI* and *DI*). Of course, under DOS, you should ensure that you use only valid 16-bit addresses in these operations. The ability to use a scaling factor of 1, 2, 4, or 8 is also valuable in addressing into arrays of different data sizes (byte, word, doubleword, or quadword.) Don't overlook the most basic advantage of being able to manipulate data in 32-bit chunks, rather than manipulating merely 16 bits at a time: this advantage offers a significant speed improvement even within the limitations of DOS's 64K segments. For example, you can add or subtract doubleword integers without register-swapping, and some direct 64-bit multiplication and division operations are available.

The instruction set of the 80386 also shows some improvements over those sets of the previous 8086/186/286 processors. Ignoring instructions that only apply to protected and virtual 8086 mode, the following instructions are immediately available for use in real mode. *MOVSX* and *MOVZX* move with sign and zero-extend; *SET* sets a byte on the same range of conditions as the J . . series (such as *SETZ, SETNZ*); you can use *MOV* to move to and from the new control, test, and debug registers (*CR0, DR0, TR6*, and so forth), *SHLD* and *SHRD* do double-precision shifts; *BSF* and *BFR* scan for a set bit; *BT, BTC, BTR*, and *BTS* copy a specified bit into the carry flag for testing, while *LFS, LGS*, and *LSS* load a full pointer into the corresponding segment register (*FS, GS*, or *SS*) and specified 16 or 32-bit destination register. As well as these new instructions, there is also a liberal range of 80286 instructions enhanced to work with 32 bits, including *IMUL*, stack-related operations *PUSHAD, POPAD, PUSHFD, POPFD*, and *IRETD*, string instructions *CMPSD, LODSD, MOVSD, SCASD, STOSD, INSD*, and *OUTSD*, and conversion instructions *CWDE* and *CDQ*.

Overall, the instruction set and its modes offer a rich and diverse range of new possibilities for programmers to take advantage of.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 60.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

80386 Protected Mode Initialization

Sometimes you can *go home again — at least when your home is the 80386's real mode.*

by Neal Margulis

The 32-bit mode of the Intel 80386 and the 80386SX provides significant architectural advantages over the 80286. In addition, software that takes advantage of these advanced features has significant performance improvements. An application program running in the 80386's native 32-bit mode typically executes from two to six times faster than the equivalent application written for the 80286. Furthermore, programs that manipulate large data structures are easier to write when you use the 32-bit mode of the 80386. Among the features that the 80386 provides over the 80286 are support for large segment sizes, 32-bit data operations, and paged memory management.

The program presented here shows how to initialize the 80386 into protected mode, how to define segments greater than 64K in size, and how to return to real mode. You can use this program as a template for coding applications that use the 80386 features. Although the 80386SX has a reduced physical addressing space of 16 Mbytes (the maximum address space of AT architecture), its programing model is the same as that of the 80386. Thus the template can be used with it as well.

This article explains how the code works and briefly describes how to adapt the template to suit your individual needs. You may also find it helpful to refer to one of many 80386 programming articles, such as "Programming on the 80386" (*DDJ*, October 1986). Additional information can be found in the *Intel 80386 Programmers Reference Guide* and the *80386 Data Sheet*, as well as the book *Programming the 80386* by Crawford and Gelsinger (Sybex Books).

Neal Margulis is an applications engineer for Intel Corp. and can be reached at 2625 Walsh Ave., SC4-40, Santa Clara, CA 95051.

32-Bit Data Operations

The ability to operate on 32 bits of data adds power to arithmetic and logical instructions. While the 80286 generates only 16-bit data, the 80386 contains eight general-purpose 32-bit registers. Segments for 80386 protected mode are set to either *use16* or *use32*, which indicates the default sizes for data and addressing. In real mode, the 80386 is limited to only *use16* segments. An override prefix must be designated in order to perform 32-bit operations within a protected mode *use16* segment. This results in greater program length and a possible decrease in performance. The 80386 in protected mode allows for both *use16* and *use32* segments. No override prefixes are necessary for 32-bit data operations or 32-bit addressing in a *use32* segment. In the program shown in Listing One, page 84, CSEG and C3 are *use16* code segments because they must be executable from real mode, and PMODE segment is a *use32* code segment.

Large Segments

The 64K limit on the segment size of the 80286 and the real mode 80386 hinder the addressing of large data

arrays and of long sequences of code. Reloading segments is time consuming, disrupts the task at hand, and causes an unnatural breakup of procedures and data. The 80386 protected mode allows for segments up to 4 gigabytes in size. The base, limit, and granularity fields of segment descriptors specify the segment size *and* location in memory. The base represents a linear address and the segment size is determined by the limit and the granularity (G) bit. When the G bit is a zero, the actual limit is the 20-bit limit field of the descriptor (Maximum size 2^{20} = 1 Mbyte). If the G bit is a one, then the limit field page granularity is multiplied by 4K. This gives a maximum limit of 4 gigabytes ($2^{20} * 2^{12}$). The base's linear address is 32-bits long, thus allowing it to be specified anywhere within the 4-gigabyte address space.

Segmentation is the basis for protection. Data and code segments can reside in separate, nonoverlapping areas of memory. In addition, privilege levels assigned to different segments provide a mechanism for limiting access to certain data or privileged instruction sequences (for both). The 80386 provides four privilege levels. In the program presented in Listing One, all segments are of the highest privilege level (0). You can change this by modifying the segment descriptors and the selectors.

4-Gigabyte Addressing

To use the increased segment sizes, the 80386 has expanded the instruction pointer to 32-bits and added new addressing modes. As a result, segment loads and stores within a procedure can be eliminated and the entire physical address space can be accessed as one segment. The template program sets up a data segment that starts at the base of video memory 0B8000H. The entire address space is accessible through a segment that begins at zero.

The effective address of a memory operand can be obtained through an absolute address or through one of the register-base methods of the following form:

```
[base register] + [(index register *
                   scale) + displacement]
```

The 80386 also has page translation by which linear addresses can be resolved to physical addresses. Page translation occurs when the PG bit in CR0 is set. Two levels of tables are used to address each page of memory. The higher-level table is the page directory, which addresses up to 1K second-level page tables. These second-level page tables address up to 1K pages, each being 4K.

Because all pages are of equal size, page translation can reduce the memory fragmentation that occurs when using segmentation for on-demand memory allocation. The template program does not use paging, so all linear addresses are treated as the physical address.

Applications programs that take advantage of the native 32-bit mode of the 80386 typically perform from two to six times faster than the same application written for the 80286

Setting Up the Descriptor Tables

Before entering protected mode, you must set up descriptor tables and load the 80386 with pointers to these tables. While in real mode, the program in Listing One sets up a global descriptor table (GDT) and does not require a local descriptor table.

Starting at the memory location designated by the label *GDT_table*, successive 8-byte descriptor entries make up the GDT. MASM's STRUC feature makes coding of the entries much easier. Because DOS determines the memory location of the program at run time, the absolute addresses must also be calculated at run time.

The template program determines the bases for each of the segments and the pointer to the descriptor table. This table pointer consists of a 32-bit linear address and a 16-bit limit.

Such 48-bit (6-byte) objects are sometimes referred to as a *PWORD* or *FWORD* data type. Using a *QWORD* (8 bytes), as in this program, helps to maintain portability between assemblers.

The Type field of the descriptor determines whether segments that use this descriptor contain code or data. Descriptors to specify gates, task state segments, and local descriptor tables are also available, but are not used in this example.

Entering Protected Mode

Having set up the descriptor table, it is simple to enter protected mode. The PE bit of Control Register Zero (CR0) is set to one, and then a jump is executed. The jump flushes the prefetch queue, which contains instructions that were decoded for execution in real mode. Either a near or a far jump will flush the prefetch queue. By using a far jump, the 80386 reloads the code segment register (CS) and the internal segment descriptor cache. The far jump instruction uses selector 08H, which is GDT entry 1. This is the PMODE segment entry. Execution in this segment allows native 32-bit operations. (Bits 15 through 3 of the selector determine the GDT table entry. Bit 2 is Table indicator, and bits 1 and 0 are the privilege level.)

The Jump instructions are hand-coded by using the define byte (DB) assembler directive. Opcode 0EAH specifies an intersegment jump, with the next two fields of the instruction being the offset and the segment selector operands. In this example, all of the offsets are zero because the jump targets are at the start of their respective segments. The offset size is either a 16-bit or 32-bit field. This is determined by the code segment type size in which the jump instruction occurs. The segment selector field of the jump instruction determines which descriptor table entry is the target segment for the jump.

Returning to Real Mode

Returning to real mode on the 80286 requires that you reset the processor. On an PC AT, this means saving the required processor contents, placing a reset code in the CMOS RAM,

storing a return address in memory, and using the keyboard controller to reset the processor. Although 386-based ATs support this reset scheme, a much simpler and far faster way to return the 386 to real mode is available. In the C3 segment of the example program, the 80386 is returned to real mode by clearing the PE bit then executing a jump instruction to flush the instruction queue. To assure proper operation after returning to real mode, the segment registers must be loaded with real mode type selectors while still in protected mode. Entry 4 in the GDT represents what should be in the segment descriptor caches during real mode. They have a 64K limit with the base at zero and the top 2 bytes set to zero.

Other Considerations

In the early days of PCs, some programs took advantage of addresses wrapping around to zero after the limit of the 8088 was exceeded. When the 80286-based AT was introduced, it was necessary to emulate this address wrapping. An enable gate was added to address line 20. To prevent unwanted wrapping, you must enable this gate. The code for doing this is located in the *IBM Technical References for the AT and PS/2*. The procedures are different because the AT uses the keyboard controller to enable the address line. The programming example indicates where to insert the procedures for enabling and disabling of the address line.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing starts on page 84.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

32-Bit Software Development

Software developers who decide to write 32-bit applications base their decisions on three primary criteria. The first criterion is performance, since 32-bit applications run two to six times faster than the equivalent 16-bit applications. The second criterion is engineering resource—32-bit software is easier to develop than 16-bit, thus requiring fewer software man-years. And finally, 32-bit applications consume less code space.

In the 32-bit 80386 programming architecture, data structures need not be broken into 64K segments, as is required the 16-bit programming model in earlier Intel processors. Nor are 32-bit applications forced to use the extended address space as LIM memory. A 32-bit application can address directly and linearly all available memory with little or no segment and page swapping, and with no restrictions on the size

32-BIT SOFTWARE

of data structures.

With the older 16-bit programming model, the "small model" program is the simplest to develop. This model consists of one code segment and one data segment. The limitation of 64K each for both code and data does cramp the application.

To overcome these size limitations, a 16-bit application can be migrated to the "large model," with multiple separate code and data segments of up to 64K each. You do face the challenge of breaking up the code and data to fit into segments. A performance penalty can result from the overhead of unnecessary procedure calls, "long pointer" arithmetic, and indexing into large data structures that have been artificially broken into 64K chunks

The 32-bit 80386 programming architecture allows you to simultaneously take this process through a giant leap backward and forward. The leap is backward in the sense that the program can parallel the small model with one large data segment and one large code segment. The leap is forward in that the application realizes a sizable increase in performance and a reduction in code size.

The 80386 programming architecture allows you to reap the benefits of segmentation without the downside of limited-size segments. The primary benefit of segmentation is transparent code relocatability. The 8086 and 80286 realized this benefit, but left you with the task of working within the confines of 64K segments. The 4-gigabyte segment size provided by the 80386 is unlikely to restrict any application for many, many years to come.

The performance gains of 32-bit applications come from several factors. The most straightforward of these factors is that 32-bit integers are simply manipulated faster. A C language integer declared "long int" will consume only one 32-bit register internally on the 80386SX or 80386 processor running in 32-bit mode, versus two 16-bit registers on the 8086 or 80286. A 32-bit register-to-register move operation occurs in one instruction on the 80386SX or 80386 versus two instructions on the 8086 or 80286.

When a 32-bit compiler generates code to multiply two long (32-bit) integers, it simply executes one multiply (MUL) instruction. The 16-bit architectures force the compiler to synthesize the 32-bit multiplication from several 16-bit instructions. Because of the length of the operation, the 16-bit compiler will execute the 32-bit multiplication as a call to a library routine and will incur an additional performance penalty. As Table 1, page 42, indicates, a multiplication of two long integers occurs in 34 clock counts on a 80386 processor executing code generated by a 32-bit compiler versus 123 clock counts on the 80386 processor executing code generated by a 16-bit compiler. Not only is the performance greater for the 32-bit version, but also the code size is smaller.

Consider a 16-bit application consisting of at least one large data structure (greater than 64K). Before moving a unit of data to or from memory, the application must check the segment register (or selector) to see if the contents are valid, and then load a new selector when appropri-

(continued from page 40)

ate. The overhead associated with managing and reloading the segment selectors decreases performance penalty. Even when programming in a high-level language, this process requires help from the programmer to break the data structure into logical partitions and to deal with the added complexity of "long pointers."

A 32-bit application consisting of the same large data structure would use simple linear addressing to move units of data directly to and from memory with no regard for segmentation. This results in optimum performance from a minimum software development investment. Additional performance gains could be realized by using the 80386's scaled indexing capability. Examples of software that clearly benefit from linear addressing are bit-mapped graphics applications (targeted at systems with large frame buffers) and database applications.

— Bruce Schechter
Neal Margulis

```
                int customers= 426000, count=219;
                sales= customers * count;

INSTRUCTION SEQUENCE                    INSTRUCTION SEQUENCE
16-bit code                             32-bit code
Clock counts = 123                      Clock count= 34

 push    WORD PTR _count+2               mov ESI, DWORD PTR _count
 push    WORD PTR _count                 mov EAX, DWORD PTR _customer
 push    WORD PTR _customers+2           mul ESI
 push    WORD PTR _customers
 call    __aNlmul
 mov     WORD PTR _sales,ax
 mov     WORD PTR _sales+2,dx


;|*** Routine  __aNlmul

 push BP
 mov BP,SP
 mov AX, WORD PTR [bp+06]   ;  {06H}
 mov BX, WORD PTR [bp+0A]   ;  {00}
 or BX,AX
 mov BX, WORD PTR [BP+08]   ;  {0DBH}
 jnz     __aNlmul+1b
 *
 *
 *

 mul BX
 mov CX, AX
 mov AX, WORD PTR [bp+0A]   ; {8010H}
 mul WORD PTR [BP+0A]       ; {00}
 add CX,AX
 mov AX, WORD PTR [BP+04]   ; {8010H}
 mul BX
 add DX, CX
 mov SP, BP
 pop BP
 ret
```

Table 1: *Example of 32-bit multiply on 80286 and 80386*

A Double Cross for MASM

Introducing Xxref, a multifile cross-reference utility for assembly-language programmers

by Steve Heller

Last year, while working for a small software house, I had the unenviable chore of working on a large assembler program that had mostly been written by a departed programmer and that was being modified by several people at *the same time*. The project consists of more than 40 files and approximately 1 Mbyte of source. Keeping track of the various changes was going to be difficult at best and presented the distinct possibility of unrestrained chaos.

One of the countermeasures I took was to write, in my own time, a utility to provide cross-references to all public symbols and their uses among the many source files that made up the program. I decided to spend my own time on this project, for two reasons. First, the company was reluctant to pay me to work on it, and second, by developing the utility in my own time, I'd eventually be able to share it with others.

After completing the utility, which I have named Xxref (cross-cross reference) to distinguish it from single-file utilities such as the one included with MASM, I decided that writing this article was the best way to make it available to those who could use it the most.

Steve Heller has been programming for more than 18 years and is the president of Chrysalis Software Corp., P.O. Box 0335, Baldwin, NY 11510 He can be reached through CompuServe: 71101,1702.

General Description

The basic method for producing the cross-reference listing is as follows:

1. Initialize the data structures, initialize the data structures, including reading in the names of all the source files to be processed.
2. Read through each file, looking for *PUBLIC* declarations and and storing the variable names and labels that are encountered in a symbol table.
3. Read through each file again, looking for references to the public variables or labels encountered during the first pass and adding these references to the linked list of references for each *PUBLIC* symbol. When finished, compact the results for sorting.
4. Sort the symbol table by symbol name.
5. Write the output either in print format, with headers and page numbers, or in screen format, with only one header at the top, as specified by the user.

Implementation

Because I wanted the utility working as soon as possible, I opted to use a high-level language (Turbo Pascal). For reference in the following discussion, refer to Listing One, which starts on page 87.

As mentioned, the program starts by initializing the various data structures (lines 614 – 699). First, it marks the heap so that it can free all the dynamically allocated storage with one statement at the end of the program. It then allocates storage for the Public Variable Array (*PublicVarArray*) and the Reference Pointer Array (*RefPtrs*).

If the user hasn't specified all the parameters on the command line, the program has to prompt for the missing ones. The first parameter is the name of the input file (which contains the names of the files to be processed), the second parameter is the name of the output file (to which the report will be written), and the third tells Xxref whether the user wants brief output (no page headers). The default is to print page headers.

Now the program can read in the names of the source files to be processed from the input file. Because I wanted to be able to accept a file produced by redirecting the output of the DIR command to a file, lines that are null, or that start with a blank or a period, or that refer to a file of zero bytes, are ignored. Otherwise, the program combines the file name and extension into one string. If the extension is null, it adds a . (period) to the name. Then it allocates storage for the file name and stores a pointer

to it in the *FileName* array.

When Xxref has read in all the file names, it closes the input file, and sets the *FileCount* variable to the number of names it has read in. Because it hasn't yet seen any public symbols, it initializes the *PublicCount* variable to 0. Then it initializes the pointers in the *PublicVarArr* array —which will point to the public symbols as it encounters them—to the *nil* pointer.

Taking a First Pass

The job of *Pass1* (lines 703 – 755) is simply to read through the files and find all the public symbols. For each file that was named in the file-name input file, it reads in each line of the file, converts it to all uppercase, changes all tabs to blanks, and removes the first token from the line. A token is something that might be an identifier because it is made up of characters that are legal in identifiers, as specified in the constant *LegalChars*. The first token in the line must be *PUBLIC* for the program to process it on this first pass, because it is just looking for all the public symbols at this point.

If *Pass1* finds the token *PUBLIC* at the beginning of the line, it removes all the rest of the tokens, one at a time, and sees whether pointers to them are already stored in the *PublicVarArr* array. If not, it adds them to that array and increments the number of *PUBLIC*s that it has found. The procedure continues extracting tokens until there is nothing left in the line. This process is continued for each line in all the files.

One More Time

In the second pass (lines 759 – 828) the program reads through all the files again, this time to find all the references. Following the second pass, it compacts the results for sorting (lines 832 – 860).

Pass2, the second pass, starts by initializing the *RefPtrs* array, which is an array of records that keeps track of the first and last references to each public symbol. It sets the *First* element of each of these records to 0, to indicate that it hasn't yet seen any symbol references. It also initializes the *RefCount* variable, which is used to index into the *RefArrs* array. *RefCount* keeps track of the number of references that the program has seen so far.

Next, for each file that was specified in the file-name input file, *Pass2* reads each line of the file and searches it for public symbols. Whenever it finds one, it increments *RefCount*.

You will notice that the *RefArrs* array is allocated in pieces rather than all at once; if *RefCount* is 1 more than a multiple of *RefSize*, then the program has to allocate another block of the *RefArrs* array. One reason for this is that no one dynamically allocated variable can be more than 64K bytes long, and the *RefArrs* records are 6 bytes long. Thus, one array of such records must contain fewer than 11,000 records. For Xxref to be able to handle 65,000 references, it must allocate the array in sections. This strategy also allows a smaller program to be analyzed with less memory usage.

After the allocation of a new section, if necessary, *Pass2* tests to see whether it has already seen a reference for the public symbol, by looking at the *First* field for the symbol. If that is zero, this is the first reference, so it sets *First* to the current *RefCount*. Otherwise, it uses the *Last* field to access the last reference on the chain and calculates the block number and item number within the block where the last reference is stored and updates its *Next* field to the number of the new reference it is constructing.

In either case, the procedure now sets the *Last* field to the number of the new reference it is constructing. Then it fills in the *FileNum*, *LineNum*, *Next*, and *Define* fields with, respectively, the number of the file in which it encountered the reference, the line number of the reference, zero (meaning there is no next reference to this symbol yet), and *FALSE*, which indicates that this is a reference and not a definition. This last is used to indicate, on the output, where the symbol was defined. Then, if it is at the beginning of a line, *Pass2* must check whether this is, in fact, a definition. If it is, the value of *LookAhead* will be *TRUE*, which will result in setting the *Define* field to *TRUE*.

Next, *Pass2* computes the block number and offset within the block where the new reference it has just constructed will be stored and copies the temporary record to its permanent location in that block.

It can now set the *StartOfLine* flag to *FALSE*, as it has processed the first token on the line. It removes the token and continues processing the same line. When no more tokens are available in the current line, it reads another line from the file. When no more lines are available from the current file, *Pass2* closes the file.

If there is another file to be read, *Pass2* opens it and processes it, as described earlier. If there are no more files, the routine finishes.

CompactResults

In order to speed up the search for symbols during pass 2, the program stored them in a hash table (see the routines *Locate* [lines 276 – 295], *Store* [lines 428 – 444], and *Hashit* [lines 252 – 271]). To sort the symbols by name, however, it must compact the table; otherwise, it will waste a lot of time sorting unused table entries. Therefore, *CompactResults* steps through the *PublicVarArr*, *PublicFileArr*, and *RefPtrs* arrays and moves all the entries that contain data to the front of the arrays, clearing out the old entries to prevent duplication.

Sorting the Symbol Table

The sort procedure *Megasort* (lines 159 – 219) is a destribution sort that I've described in detail in another article[1], so I will not describe its operation here. For the principles of operation, you can refer either to my article or to Donald Knuth's classic *The Art of Computer Programming*.[2] In short, the merits of a distribution sort are these: It's fast (for small keys), it's easy to implement, and the performance isn't data dependent.

The arguments to the sort procedure are, in order, the array of pointers to the keys to be sorted (the symbol names), an array to be rearranged along with the key pointers (the file numbers), another array to be rearranged *along* with the key pointers (the reference pointers), the number of bytes of the keys that are to be considered significant, and the number of keys to be sorted. Upon return from the sort, the key pointers are reordered by the sorted values of the keys, and the other two arrays

are reordered along with the keys that they are associated with.

Getting the Results

The *WriteResults* procedure (lines 864 – 936), generates output either in print format, with headers and page numbers, or in screen format, with only one header at the top, as specified by the user.

Both brief output (no headers or footers and no page breaks) and listing output (headers, footers and page breaks) are supported by the *WriteResults* routine. The brief mode is preferable if the cross-reference is to be used on-line in an editor as headers on each page are unnecessary in such a use. The listing mode is better for printed output as it is harder to scroll back and forth in a printed listing and compactness is not as important. For an example of the listing output, see Example 1, page 50. The brief output is basically the same, except that the headers are not repeated and no page breaks are inserted in the output.

In order to provide headers and footers on each page of the output, the program must keep track of the number of lines that have been printed. For simplicity in the main output routine, this chore has been relegated to the *WriteOut* routines (*WriteOutCr*, *WriteOut*, *WriteOutLn*, and *WriteOutSym*), which replace the normal *Write* and *WriteLn* routines. Each of these *WriteOut* routines includes the check for end of page and calls the *Header* routine to take care of the printing of the footer and header, if needed. The *Header* routine can also handle the breaking of a listing of one symbol over a page boundary by adding (*Continued*) to the listing on the new page.

The merits of a distribution sort are these: It's fast, it's easy to implement, and the performance isn't data dependent

After the initial header has been printed, which occurs even if the output is being done in brief mode, *WriteResults* starts processing all symbol references.

For each symbol, the first line of output includes its name, a line of periods, and the name of the file in which it was defined. If no references to that symbol were found, then a

```
        XXREF V1.0 - Copyright 1987 by Chrysalis Software Corp.        Page 5

         Cross reference as of 3/13/1988 11:55:00 among files:

  ADJMENU.ASM     CMDS.ASM    COLOR.ASM  DATASEG.INC   DISKIO.ASM     DRAW.ASM
     END.ASM    ERROR.ASM  FGLOBAL.ASM       FL.ASM    FLASH.ASM  FLASHUP.ASM
   FLNBOX.ASM  FLNOTES.ASM   FMACRO.ASM FUNCTION.ASM   FWCLIP.ASM     FWED.ASM
   FWLINK.ASM   FWMENU.ASM   FWSUBS.ASM  GETITEM.ASM   GLOBAL.INC     HARD.ASM
    HORIZ.ASM      INTS.ASM    KEYIO.INC   LOADER.ASM   MEGAFU.ASM     MENU.ASM
 MENUDATA.INC MOUSECTL.ASM MOUSEKEY.ASM     MOVE.ASM MVERSION.INC  OVERLAP.ASM
  READBOX.ASM   RETSTR.ASM      SAL.ASM     SCAN.ASM  SCRMACS.INC SCRPROCS.ASM
   SEARCH.ASM SHOWHELP.ASM     SUBS.ASM       TP.ASM    VIDEO.ASM

Public symbol                    Declared in file
-------------                    ----------------

@FPRINT_STR.....................(Continued)

     FWMENU.ASM        43
     FWSUBS.ASM        55    372    731    869    886   1622


@FPRINT_STR_NOCOLOR.............FL.ASM

     DRAW.ASM          22
     FL.ASM            23*    30
     FWMENU.ASM        44    614
     FWSUBS.ASM        56   1397
```

(continued on page 52)

Example 1: *A sample page of Xxref's output*

DOUBLE CROSS
(continued from page 50)

blank line follows. Otherwise, the routine prints the name of each file in which a reference was found, followed by the line numbers of all references—a maximum of eight per line. Any line on which the symbol is defined will have an * (asterisk) next to the line number. Whenever a reference comes from a different file from that of the last reference, the routine starts a new line. When all references to a symbol have been printed, it steps to the next symbol after printing two blank lines.

When all references to all symbols have been printed, *WriteResults* prints the total number of public symbols and the total number of references.

Example 1: *(continued from page 50)*

```
@FPUT_CHAR.....................FL.ASM

     COLOR.ASM        11    218    221    233    235
     DRAW.ASM         22    155
     FL.ASM           67*    75
     FWED.ASM        589    612    675
     FWMENU.ASM       44    606    659    668    684    688    696    705
     FWSUBS.ASM       56   1125
     HORIZ.ASM        75    370    375    377    384    385    392    395
                     402    408


@FPUT_COLOR....................FL.ASM

     COLOR.ASM        14     82     94    109    184
     DRAW.ASM         21
     FL.ASM          133*   155
     FWED.ASM       1955   2305
     FWMENU.ASM       43    517    532    749
     FWSUBS.ASM       55
     HORIZ.ASM        75    282    286


@FPUT_CX_CHARS.................FL.ASM

     DRAW.ASM         21
     FL.ASM           71    102*   129
     FWED.ASM       2456   2474   2577

                (* = line where symbol is defined)
```

Conclusion

If you have large MASM programs to maintain, this utility should be useful. The source code for the original Turbo Pascal version described in this article is available from *DDJ* in the usual ways (diskette or CompuServe). Running times for the approximately 1-Mbyte program mentioned at the beginning of the article, which contained 1,070 public symbols with 9,290 references, were 560 seconds for the Turbo Pascal version and 160 seconds for the assembly-language version. (Both trials were timed on a 10-MHz, no-wait-state, 80286 AT clone.) This is a shareware program. If you find it useful you should register it by sending a check for $39.95, payable to Chrysalis Software Corp., to the address at the beginning of this article. Registered users will receive a copy of an assembly-enhanced version of the program.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Notes

1. Steve Heller "MegaSort: A Distribution Sort," *Computer Language* (November 1987): 63-68.
2. Donald E. Knuth. *The Art of Computer Programming*, Volume 3 (Reading, Mass.: Addison-Wesley, 1973.)

DDD

(Listing begins on page 87.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Turbo Prolog 2.0

Borland's Turbo Prolog 2.0 sports a customizable interface and increased compatibility with standard Prolog

by Ernest Tello

The release of Turbo Prolog 2.0 has made clear Borland's intention to continue improvement of this product along two distinct directions: to make Turbo Prolog more suitable to conventional business applications and to bridge the gap between it and standard Prolog. Some major new additions are a new user environment, expanded internal databases, external Prolog databases, a new graphics interface, and a near-to standard Prolog interpreter.

The User Environment

What you notice first about Turbo Prolog 2.0 is the difference in the user interface. The function key assignments have been changed to some extent. Two convenient new features are the Window zoom key (F5) and the Next window function key (F6) which enables you to cycle around the windows in a predetermined order. The most drastic change is that the whole interface is completely redefinable. And finally, the key assignment screens allow you to assign two different key commands to every function.

The Borland-style interface gives the appearance of two editors being copresent. You can use the regular editor for one module of your program, and then call up the auxiliary editor to write documentation or to work on other modules. Although this provides some of the advantages of having two simultaneous editors, in actuality the same code is being executed by both. This is apparent when you try to use the Next key to get from one editing window to another, but you can't. The main advantage here is the convenience of keeping one file loaded in the main editor while retaining the ability to work on other files without unloading the auxiliary editor.

Ernie Tello is a consultant, lecturer, and writer in the field of artificial intelligence. He can be reached at 1518 W. Cliff Dr., Santa Cruz, CA 95060.

Internal and External Databases

Two important changes have been made in the Turbo Prolog database system. The internal databases now allow multiple-named internal databases in a single application. A new type of Prolog database, a database on disk that supports full random access, can now be created. The option of creating multiple-named databases provides an interesting form of data partitioning in Turbo Prolog. To support this feature, the assertion and retraction functions have two different forms: operating as before, to support the old syntax; and with an additional argument, the name of the database.

When a named internal database is declared, you just declare the database predicates. Facts can be added to an internal database in four different ways: asserting them in the clauses section of a program, asserting them at run time, loading them from a disk file, and by using the *readterm* predicate (explained later). The *save* and *consult* predicates can now take an additional argument which is the name of one of the internal databases.

The expanded internal databases have some potentially powerful uses. These databases can be used as a form of partitioned working memory to solve complex problems. This feature adds the potential to write programs that operate like forward chaining systems. This means that assertion and retraction can occur in powerful ways within distinct memory areas.

Another important advantage of the multiple databases is the ability to do complex disk operations interactively while programs are running. The fact that files may be saved and loaded into named data areas provides an important form of modularity that allows for creative application development. For example, a program can be written to read disk files into named database areas, perform inferences that result in new assertions added, and then save the expanded files to disk. Each time the program is run, it can be operating with more expanded databases that enable additional inferences to be made.

This type of programming is much different from the pure Prolog advocated in many of the traditional Prolog programming texts. According to the traditional style of programming, assertion and retraction is performed sparingly. Turbo Prolog is far from being the purists type of Prolog. Although it is a Prolog with idiosyncrasies, compromises, and tradeoffs, the style of programming that can work effectively in Turbo Prolog is not the same as for standard Prolog.

Turbo Prolog provides a means to keep internal databases on disk. The documentation shows how user-defined predicates can be written that do this and even how to use index files to record the position of facts in the data file.

The addition of external database support to Turbo Prolog means that the size of applications is no longer confined to the space allowed by internal memory. One of the main limitations of Turbo Prolog has been the size of programs that can be written. With the external database capability, valuable memory no longer need be used for storing factual data but can now be done with random-access files. These databases are not

the usual kind of field and key databases but rather Prolog databases.

A main difference between internal and external databases is the facilities that external databases provide. External databases are stored in binary format in chains of like terms, and B-trees can be used when the databases need to be sorted. Access is provided to external databases through database reference numbers. Predicates are provided that can manipulate whole databases, chains, and terms. The real difference between internal and external databases, therefore, is the type of access available for each.

A useful way of combining internal and external databases might be to use internal databases for storing results obtained by querying external databases. At first the result stored there can be temporary. But by making other inferences, further assertions of a more permanent type can be made into a different internal database which can be saved to disk if desired. External databases are more like conventional databases; they have many accessing functions and features that make large systems of records maintainable. Once you become familiar with using the two types of Turbo Prolog databases, various techniques will allow you to design the optimal mix of the two for given applications.

Borland has introduced its BGI (Borland Graphics Interface) into Turbo Prolog. Previous versions of the language had graphics but not on this level of sophistication. Turbo Prolog now has 70 graphics-related predicates. This graphics system includes support for defining viewports, which are windows in the graphics mode. As with text-mode windows, viewports show only the part of a graphic image that falls within its borders. Perhaps the nicest part of the graphics interface are the drawing predicates that are provided. With *setlinestyle* you can specify the thickness and visual style of the lines drawn. Built-in predicates are included for drawing shapes such as circles, ellipses, rectangles, polygons, arcs, and pie slices. For business purposes, predicates are provided for drawing bar charts (including three-dimensional bar charts). Predicates are also supplied for performing various types of fills.

The Interpreter

Borland's Prolog interpreter comes closer to supporting standard Prolog than the compiled language has up to now. None of the I/O and debugging predicates of standard Prolog are supported, since Turbo Prolog already has its own. A surprising number of the standard features are present, including those for AI purposes. The interpreter comes in Turbo Prolog source code and is run like any other application in Turbo Prolog.

The interpreter comes with its own built-in editor that works essentially the same way as the main editor in the development environment. One important difference is that if a file has been loaded into the interpreter, it automatically appears in the editor when the editor is called up. This handy feature means that you can conduct an interactive session, call up the editor at any time, and inspect and edit the current contents of Prolog memory to save it to disk.

The Prolog interpreter included as a program in this version is closer to Clocksin and Mellish than it is to Turbo Prolog. Since the dialect is different from Turbo Prolog, working with the two different dialects could be more trouble than it's worth.

I don't agree with that assumption. I find myself applying at least two rather important uses to this interpreter. One is trying out ideas that I will later import into the compiled dialect. The advantage here is that you don't have to make type declarations first but can start coding right away without any preliminaries. This is not for large programs, mind you, but rather for small experimental "laboratories" for ideas. With the compiled language, you're better off making type declarations as you go along, rather than all at once.

The second use of the interpreter is for writing things that you cannot write in compiled Turbo Prolog. If need be, you can even try to embed the interpreter in an application to run that code. With this release of the interpreter, the tradeoff between I/O and the need for meta-predicates and run-time binding has been overleaped to some degree.

Interfacing to Other Languages

An important addition with this version is support for a two-way interface with outside languages. Previously you could call C functions from Turbo Prolog if those functions followed the rules. Now you can do the opposite as well. Predicates can be written and compiled and then called from C programs. The advantage here is that you can write an advanced database system in Turbo Prolog and then embed this in a program written in another language. You can, thus, take advantage of something not available to you in Turbo Prolog (such as some code that you or another developer might have already written).

Whither Turbo Prolog?

Those programmers who have applications written in the old version of Turbo Prolog should have no problem making them run in version 2.0. I tested a large program written in the original Turbo Prolog, and it ran without any problems.

The main limitation of the language now is the size of programs that can be written. It's now becoming state-of-the-art for programming languages to take advantage of the protected mode of the Intel 286 and 386 processor series. It is only natural to expect that Borland will wish to stay in pace with the industry and produce a protected-mode version of Turbo Prolog that will be able to run in the extended memory areas of 16 Mbytes and higher.

Another important direction that Borland could take is to make the compiler support the version of Prolog implemented in the interpreter. Those programmers who are interested in the features of Prolog that make it suitable for AI applications would be able to enjoy the fast running speed of compiled Turbo Prolog code, as well as to take advantage of the wide range of I/O capabilities provided in version 2.0 and in the Toolbox. Combined with the ability to run in protected mode, these would represent an unbeatable combination of features for a Prolog implementation.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

EXTENSIONS TO LISP

Listing One *(Text begins on page 18.)*

```
(defmacro for (var-from-to &rest body)
  (let ((var (first var-from-to))
        (from (second var-from-to))
        (to (third var-from-to)))
    `(prog (,var)
           (setq ,var ,from)
      loop
           (cond ((> ,var ,to) (go end)))
           ,@body
           (setq ,var (+ ,var 1))
           (go loop)
      end)))
```

End Listing One

Listing Two

```
(defmacro for (var-from-to &rest body)
  (let ((var (first var-from-to))
        (from (second var-from-to))
        (to (third var-from-to)))
    (cond
     ((and (numberp from) (numberp to) (< (- to from) 2))
      ;; If from and to are both numbers, and they differ by at most 1..
      (cond ((< (- to from) 0)
             ;; they differ by < 0, hence there's no loop to generate
             nil)
            ((= (- to from) 0)
             ;; they're the same, so just a single iteration
             `(let ((,var ,from))
                ,@body))
            (t
             ;; else, they differ by one: so two iterations
             `(let ((,var ,from))
                ,@body
                (setq ,var ,to)
                ,@body))))
     (t ;; the general case
      `(prog (,var)
             (setq ,var ,from)
        loop
             (cond ((> ,var ,to) (go end)))
             ,@body
             (setq ,var (+ ,var 1))
             (go loop)
        end)))))
```

End Listing Two

Listing Three

```
(defmacro for (clause &rest body)
  (let* ((code (funcall (get (second clause) 'for-expander)
                        (first clause) (cddr clause)))
         (init (first code))
         (test (second code))
         (update (third code)))
    `(prog ()
           ,@init
      loop
           (cond (,test (go end)))
           ,@body
           ,@update
           (go loop)
      end)))
```

End Listing Three

Listing Four *(Text begins on page 18.)*

```
(defmacro for (&rest forms)
  (let* ((do-part (member 'do forms))
         (body (cdr do-part))
         (clauses (ldiff forms do-part)) ;clauses = everything before "do"
         (init nil)
         (test nil)
         (update nil))
    (dolist (clause clauses)
      (let ((code (funcall (get (second clause) 'for-expander)
                           (first clause) (cddr clause))))
        (setq init (append init (first code)))
        (push (second code) test)
        (setq update (append update (third code)))))
    (setq test (cons 'or (nreverse test)))
    `(prog ()
           ,@init
      loop
           (cond (,test (go end)))
           ,@body
           ,@update
           (go loop)
      end)))
```

End Listing Four

Listing Five

```
(defmacro defrecord (name &rest components)
  `(progn
     ,@(accessor-macro-defs name components)
     (defun ,(symbol-append 'make- name) ,components
       (let ((new-record (make-array ,(length components))))
         ,@(component-setting-list name components)
         new-record))))

(defun component-setting-list (name components)
  (let ((set-list nil))
    (for (comp in components)
         do
         (push `(setf (,(accessor-name name comp) new-record) ,comp)
               set-list))
    set-list))

(defun accessor-macro-defs (name components)
  (let ((def-list nil))
    (for (i from 0 to (- (length components) 1))
         do
         (push `(defmacro ,(accessor-name name (nth i components)) (x)
                  (list 'aref x ,i))
               def-list))
    def-list))

(defun symbol-append (&rest symbols)
  (intern (apply #'string-append symbols)))

(defun accessor-name (rec-name comp-name)
  (symbol-append rec-name '- comp-name))
```

End Listings

FORTH 386 ASSEMBLER

Listing One *(Text begins on page 28.)*

```
SCREEN 0

\ 80386 Assembler                                        10jul88 JBD
\
\                    80386 Assembler
\            Copyright (c) 1988 by John B. Dilworth
\
\ Permission is given to freely use or distribute this program,
\ provided that this entire copyright notice is included on all
\ copies of the source code, or any documentation of the
\ object code.  It is released as 'Shareware'; please remit
\ an appropriate amount, based on usage, for registration,
\ extra documentation, updates, etc., to the author at
\ 133 N. Arlington St., Kalamazoo, MI 49007.
\
\ My thanks to Mike Perry and Henry Laxen, whose public-domain
\  F83 Forth system and 8086 Assembler inspired this program.
\
SCREEN 1

\ 80386 Assembler Load Screen                            10jul88 JBD

: FAF  ONLY FORTH ALSO ASSEMBLER ALSO FORTH ;  FAF
: CODE  CODE FAF ;
\ Above to load on top of F83, 80386 assembler use only.
\ For full use of system, recompile F83 and replace 8086
\ assembler, or set up new Vocabulary for this assembler and
\ redefine CODE etc. to refer to it.

DECIMAL
2 59  THRU  CR .( 80386 Assembler loaded.)
EXIT

SCREEN 2

\ 80386 Assembler   Register, Mode Definitions           10jul88 JBD
OCTAL ( default base)
: REG    ( mode reg# -- )  11 * SWAP 1000 * OR CONSTANT   ;
: REGS   ( n mode -- )   SWAP 0 DO  DUP I REG  LOOP  DROP ;
10 0 REGS   AL  CL  DL  BL  AH  CH  DH  BH
10 1 REGS   AX  CX  DX  BX  SP  BP  SI  DI
10 2 REGS   [BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] [BP] [BX]
 4 2 REGS   [SI+BX] [DI+BX] [SI+BP] [DI+BP]
 6 3 REGS   ES  CS  SS  DS  FS GS
 3 4 REGS   #   #)  S#)
10 5 REGS   EAX ECX EDX EBX ESP EBP ESI EDI

: MD   CREATE  1000 * ,  DOES>  @ SWAP 7000 AND = 0<>  ;

0 MD R8?   1 MD R16?   2 MD MEM?   3 MD SEG?
5 MD R32?  6 MD MMI32?  7 MD MEM32?

SCREEN 3

\ DOUBLE, SIZE32, DOUBLE?, REG32, REGS32, SWD, DWD   10jul88 JBD
VARIABLE DOUBLE   VARIABLE SIZE32
: DOUBLE?  DOUBLE @ 0<> ; ( test for 32-bit displacement)

: REG32   ( mode reg# -- )
   11 * SWAP 1000 * OR CREATE  ,
   DOES>  @  ( -- constant )  DPL @ -1 = NOT  ( double word?)
    IF  -1 DOUBLE !  -1 DPL !  ELSE 0 DOUBLE ! THEN ;
: REGS32 ( n mode -- )
   SWAP 0 DO DUP I REG32 LOOP DROP ;
10 7 REGS32   [EAX] [ECX] [EDX] [EBX] [ESP] [EBP] [ESI] [EDI]

: SWD ( 16-bit disp; use to define single-word vars )
   CREATE , DOES>   SIZE ON  SIZE32 OFF ;
: DWD ( 16-bit disp; use to define double-word vars )
   SWAP CREATE , , DOES>  SIZE32 ON ;

SCREEN 4

\ IREG/S, initializing MMI32 regs [EAX+EBX] etc.         10jul88 JBD

: IREG ( base, loop index -- )
   OR 6000 OR  CREATE ,  DOES>  @ ( constant)
   DPL @ -1 = NOT ( double word?)
```

```
    IF  -1 DOUBLE !  -1 DPL !  ELSE 0 DOUBLE ! THEN ;

: IREGS ( reg# -- ) ( Init's [EAX+EBX] etc. in SIB format )
   10 *  10 0 DO  DUP  I   IREG  LOOP DROP ;

VARIABLE SPT  ( used in prefix tests)
```

SCREEN 5

```
\ IREGS, initializing MMI32 regs [EAX+EBX] etc.      10jul88 JBD
0 IREGS [EAX+EAX] [ECX+EAX] [EDX+EAX] [EBX+EAX]
        [ESP+EAX] [EBP+EAX] [ESI+EAX] [EDI+EAX]
1 IREGS [EAX+ECX] [ECX+ECX] [EDX+ECX] [EBX+ECX]
        [ESP+ECX] [EBP+ECX] [ESI+ECX] [EDI+ECX]
2 IREGS [EAX+EDX] [ECX+EDX] [EDX+EDX] [EBX+EDX]
        [ESP+EDX] [EBP+EDX] [ESI+EDX] [EDI+EDX]
3 IREGS [EAX+EBX] [ECX+EBX] [EDX+EBX] [EBX+EBX]
        [ESP+EBX] [EBP+EBX] [ESI+EBX] [EDI+EBX]
          ( ESP can't be index register)
5 IREGS [EAX+EBP] [ECX+EBP] [EDX+EBP] [EBX+EBP]
        [ESP+EBP] [EBP+EBP] [ESI+EBP] [EDI+EBP]
6 IREGS [EAX+ESI] [ECX+ESI] [EDX+ESI] [EBX+ESI]
        [ESP+ESI] [EBP+ESI] [ESI+ESI] [EDI+ESI]
7 IREGS [EAX+EDI] [ECX+EDI] [EDX+EDI] [EBX+EDI]
        [ESP+EDI] [EBP+EDI] [ESI+EDI] [EDI+EDI]
```

SCREEN 6

```
\ SREG, etc.  ( Special registers )                  10jul88 JBD

: SREG ( 13-15th bits + regval -- )  CONSTANT ;

HEX  2000 SREG CR0   2012 SREG CR2   201B SREG CR3
     4000 SREG DR0   4009 SREG DR1   4012 SREG DR2
     401B SREG DR3   4036 SREG DR6   403F SREG DR7
     8036 SREG TR6   803F SREG TR7
OCTAL

: CTL? 20000 AND 0<> ;   ( tests for control, debug, test regs)
: DBG? 40000 AND 0<> ;
: TRG? 100000 AND 0<> ;

: SPL? 160000 AND  0<> ; ( One of bits 13-15 set?)
```

SCREEN 7

```
\ Constants, Address modes, Immediate data + tests   10jul88 JBD

: D#   4033  -1 DPL !  ;              ( 32-bit immed. data)
: D#)  4050   0 DOUBLE !  -1 DPL ! ;  ( 32-bit direct mem. disp)
: SD#) 4060   0 DOUBLE !  -1 DPL ! ;
                          ( non-relative 32-bit call/jmp disp)

10000 CONSTANT *1  10100 CONSTANT *2  ( scaling factors)
10200 CONSTANT *4  10300 CONSTANT *8

: #?    # = 0<> ;
: D#?  D# =  0<>  ;

BP CONSTANT RP    [BP] CONSTANT [RP]    ( RETURN STACK POINTER )
SI CONSTANT IP    [SI] CONSTANT [IP]    ( INTERPRETER POINTER )
BX CONSTANT W     [BX] CONSTANT [W]     ( WORKING REGISTER )
```

SCREEN 8

```
\ Addressing Modes, etc.                             10jul88 JBD
: REG?    ( n -- f )   DUP 17000 AND 2000 <
   IF ( If 8 or 16-bit reg) DROP -1   ELSE R32? THEN ;

: BIG?   ( n -- f )   ABS -400 AND 0<>  ;
: RLOW   ( n1 -- n2 )     7 AND ;
: RMID   ( n1 -- n2 )   70 AND ;
VARIABLE SIZE   SIZE ON
: BYTE   ( -- )   SIZE OFF ;
: OP,    ( n op -- )   OR C,  ;

: ,/C,   ( n f -- )   IF  , ELSE  C,  THEN  ;
: RR,    ( mr1 mr2 -- )   RMID SWAP RLOW OR 300 OP,  ;

VARIABLE LOGICAL
: B/L?   ( n -- f )   BIG? LOGICAL @ OR  ;
```

SCREEN 9

```
\ Direct or Indirect Memory (Address size) tests     10jul88 JBD
: #)?  #) = 0<> ;
: D#)?  4050  = 0<> ;
: SD#)? 4060  = 0<> ;
: U#)?  DUP  #)?  SWAP D#)? OR 0<> ;

: SIZE32?   SIZE32  @ 0<> ;
: *? DUP *1 = 1 PICK *2 = OR  1 PICK *4 = OR  ( --reg, flg)
    1 PICK *8 = OR  0<> SWAP DROP ;

: UMEM32? DUP MEM32? SWAP MMI32? OR 0<> ;
: UMEM?   DUP DUP UMEM32? SWAP MEM?
   2 PICK *? OR OR 0<> SWAP DROP  ;

: UMEMA? DUP UMEM? SWAP U#)? OR 0<> ;  ( Any-memory test)
```

SCREEN 10

```
\ 32-bit operation words                             10jul88 JBD
VARIABLE USE   USE OFF
: USE?  USE @ 0<> ;
: USE16  USE OFF  SIZE32 OFF  ; ( 386 default segment types)
: USE32  USE ON   SIZE32 ON   ;
: WRAP  USE? IF ( 32-bit) SIZE32 ON ELSE SIZE32 OFF THEN
   SIZE ON ;
( Operand sizes )
: BY     ( -- )   BYTE ;
: WD     ( -- )   SIZE ON SIZE32 OFF ;
: DW     ( -- )   SIZE32 ON  ;

: W,   ( op mr -- )
   DUP R16? 1 AND  SWAP  R32? 1 AND  OR  OP,  ;
: SIZE,   ( op -- op' )
   SIZE @ 1 AND   SIZE32 @ 1 AND  OR  OP,  ;
```

SCREEN 11

```
\ MMI32*, ( disp [EAX+EBX] *x cases)                 10jul88 JBD

: MMI32*,   ( disp mr *x rmid -- )  ( mr of [eax+ebx] form)
   DOUBLE? NOT
   IF    ( test for |--|---|101|)   2 PICK
    7 AND 5 = 4 PICK 0= AND ( is it 0 [ebp+reg] case?)
    IF ( --disp mr *x rmid) 104 OP, OR  C,  C,
    ELSE ( any other case; mode 0, 1 or 2) 3 PICK BIG?
     IF  204 OP,  OR  C,  ,  0  ,
     ELSE  3 PICK 0=  ( mode 0?)
      IF ( --disp mr *x rmid) 4 OP, OR C, DROP ( mode 0,no disp)
      ELSE  104 OP, OR C, C,  ( mode 1, byte disp) THEN THEN THEN
   ELSE  ( double)  ( --disp mr *x rmid) 204 OP, ( --disp mr *x)
    OR  C, SWAP , ,  ( mode 2, 32-bit disp)   THEN ;
```

SCREEN 12

```
\ MEM*, MEM32*;  MEM32 scaling cases, disp [eax] *x  10jul88 JBD
( disp [EAX] *X cases: must code as [disp32+{scale*index}] )

: MEM32*,   ( disp mr *x rmid -- )  ( mr of [eax] form)
   4 OP, ( disp mr *x)
   SWAP ( disp *x mr)  OR ( disp rslt)
   5 OP, ( disp)
   DOUBLE?  IF  SWAP , ,  ELSE  , 0 ,  THEN  ;

 : MEM*, ( disp mr *x rmid -- )
    RMID SWAP 377 AND ( --disp mr rmid *x )  ( 8 bits only)
    SWAP 2 PICK  ( --disp mr *x rmid mr)  MEM32?
    IF   ROT  RMID  -ROT  ( __disp mr *x rmid)   MEM32*,
    ELSE  ROT  377  AND  -ROT   MMI32*,   THEN ;
```

SCREEN 13

```
\ MEM#), MEM16, ( all drct mem + 16-bit indrct mem ) 10jul88 JBD

: MEM#),   ( disp mr rmid -- )  OVER #) =   ( direct mem opnd)
   IF  RMID 6 OP, DROP ,  ELSE
    OVER D#)?  IF  RMID 5 OP, DROP SWAP , , THEN THEN  ;

: MEM16,   ( disp mr rmid -- ) ( Original indirect mem cases)
   RMID OVER RLOW OR -ROT [BP] = OVER 0= AND
    IF  SWAP 100 OP, C,  ELSE  SWAP OVER BIG?
     IF  200 OP, ,  ELSE  OVER 0=
      IF  C, DROP  ELSE  100 OP, C,
    THEN THEN THEN  ;
```

SCREEN 14

```
\ MMI32, ( disp [EAX+EBX] cases, MMI32? test)        10jul88 JBD
   ( Extra SIB byte needed)

: MMI32,   ( disp mr rmid -- )  ( mr of [eax+ebx] form)
   RMID  DOUBLE? NOT
   IF ( all non-double-disp cases)  OVER ( --disp mr rmid mr)
    7 AND 5 = 3 PICK 0= AND ( is it 0 [ebp+reg] case?)
     IF ( --disp mr rmid) 104 OP,  C, C,
                ( |01|reg|100|, =[ebp+{scl*indx}+dsp8])
     ELSE  2 PICK BIG? ( > 8 bits? If so, mode 2, 32-bit disp)
      IF 204 OP, C,  ,  0  ,
      ELSE  2 PICK 0=
       IF ( --disp mr rmid) 4 OP, C, DROP ( mode 0, no disp)
       ELSE  104 OP, C, C, ( mode 1, byte disp)  THEN THEN THEN
    ELSE ( double)  204 OP, C, SWAP  , ,   THEN ;
```

SCREEN 15

```
\ Addressing: MEM32, (disp [EAX] cases, MEM32? test) 10jul88 JBD

: MEM32,   ( disp mr mr -- )
   RMID OVER RLOW OR    DOUBLE?  NOT
   IF ( all single-disp cases)  -ROT   ( -- rslt disp opnd)
    [EBP] = OVER 0= AND
     IF  ( --rslt, disp) SWAP 100 OP, C, ( mode 1 with 0 disp)
     ELSE  SWAP OVER BIG?    ( larger than 8 bits?)
      IF   200 OP,  ,  0 ,  ( 16-bit case)
      ELSE  OVER 0=
       IF ( --disp, rslt)  C, DROP   ( mode 0, no disp)
       ELSE  100 OP, C, ( mode 1, byte disp) THEN THEN THEN
    ELSE  ( double disp) NIP  200 OP,  SWAP  ,  ,  THEN ;
```

SCREEN 16

```
\ Addressing: MEM, ( cases satisfying UMEMA? test)   10jul88 JBD

: MEM,    ( disp ?op mr mr -- )
    OVER U#)? IF MEM#), ELSE
     OVER MEM? IF MEM16, ELSE
      OVER MEM32? IF MEM32, ELSE
       OVER MMI32? IF MMI32, ELSE
        MEM*, THEN THEN THEN THEN ;
```

SCREEN 17

```
\ Segment and Segment Override handling              10jul88 JBD
HEX  VARIABLE INTER
: FAR     ( -- )    INTER ON  ;
: ?FAR    ( n1 -- n2 )    INTER @ IF    8 OR  THEN  INTER OFF ;
VARIABLE SOVROP    VARIABLE SOVRFLG    SOVRFLG OFF
: SEGOVR   ( opcode -- ) CREATE C, DOES>
    C@  SOVROP  C!  SOVRFLG  ON  ;
 2E SEGOVR CS:  3E SEGOVR DS:  26 SEGOVR ES:
 36 SEGOVR SS:  64 SEGOVR FS:  65 SEGOVR GS:
: SOVR? SOVRFLG @ 0<> ;
: SEGOVR?  SOVR? IF SOVROP C@ C,  SOVRFLG OFF THEN  ;  OCTAL

: SEG16? ( -- f; is it ES,CS,SS or DS?)    ( 12MI use)
   DUP SEG? IF 40 AND 0= ELSE DROP 0 THEN ;
: SEG32? ( -- f; is it FS or GS?)          ( 12MI use)
   DUP SEG? IF 40 AND 0<> ELSE DROP 0 THEN ;
```

SCREEN 18

```
\ Address Prefix handling: APREFX32 etc.             10jul88 JBD
( Handle Adr/Operand-Size 386 Prefixes)

VARIABLE OPSET ( 0 for 1-operand opcodes, 1 for others)
: OPSET? OPSET @ 0<> ;
VARIABLE DUN  0 DUN !    : DUN? DUN @ 0<> ;

: APREFX32 ( ...-...)
   SPT @ PICK DUP DUP  D#)?  SWAP UMEM32? OR SWAP *?  OR
    IF   1 DUN !
    ELSE  SPT @ PICK DUP  DUP MEM? SWAP #)? OR SWAP S#) = OR
      IF 147 C, 1 DUN !  THEN  THEN
    SPT @ 0=   ( move ptr to begn of source opnds)
    OPSET?  AND
     IF ( must be reg ) 1 SPT !  THEN  ;
```

SCREEN 19

```
\ Address Prefix handling: APREFX16 etc.             10jul88 JBD

: APREFX16 ( ...-...) ( USE16, 32 bit adr. cases)
   SPT @ PICK DUP  #)?  SWAP MEM?  OR
    IF   1 DUN !    ( no adr-size prefix reqd)
    ELSE  SPT @     PICK DUP  UMEM32? SWAP D#)?   OR ( --..flg)
          SPT @ 1+ PICK DUP  *?      SWAP SD#)?  OR  OR
     IF 147 C, 1 DUN !  THEN  THEN
   SPT @ 0= ( move ptr to begn of sOs opnds)
   OPSET?  AND
    IF ( must be reg) 1 SPT ! THEN ;
```

SCREEN 20

```
\ Operand Prefix handling: OPREFX32                  10jul88 JBD

: OPREFX32 ( ...--...)  ( USE32, but 16-bit opnds?)
   SPT @ PICK  DUP  D#?  SWAP R32?  OR
    IF   1 DUN !
    ELSE  SPT @ PICK DUP  R16? SWAP #?  OR
     IF 146 C, 1 DUN !  THEN  THEN
   SPT @ 0=   ( move ptr to begn of source opnds)
   OPSET?  AND
    IF DUP  REG? IF 1 SPT !  THEN
     DUP DUP  MEM?  SWAP #)?  OR  IF 2 SPT !  THEN
     DUP D#)?  IF 3 SPT !  THEN
     DUP UMEM32? IF DOUBLE? IF 3 SPT ! ELSE 2 SPT ! THEN THEN
     DUP   *?   IF DOUBLE? IF 4 SPT ! ELSE 3 SPT ! THEN THEN
    THEN ;
```

SCREEN 21

```
\ Operand Prefix handling: OPREFX16                  10jul88 JBD

: OPREFX16 ( ...--...) ( USE16,  but 32 bit operands?)
   SPT @ PICK DUP  #?  SWAP R16?  OR
    IF   1 DUN !   ( no opnd-size prefix required)
     ELSE  SPT @ PICK DUP  R32?  SWAP D#?  OR
    IF 146 C, 1 DUN !  THEN  THEN
   SPT @ 0= ( move ptr to begn of source opnds)
   OPSET?  AND
    IF DUP  REG? IF 1 SPT !  THEN
     DUP DUP  MEM?  SWAP #)?  OR  IF 2 SPT !  THEN
     DUP D#)?  IF 3 SPT !  THEN
     DUP UMEM32? IF DOUBLE? IF 3 SPT ! ELSE 2 SPT ! THEN THEN
     DUP   *?   IF DOUBLE? IF 4 SPT ! ELSE 3 SPT ! THEN THEN
    THEN ;
```

SCREEN 22

```
\ Addrs/Operand Prefixes: PREFX, ADRPREFX, OPNDPREFX 10jul88 JBD
: OPNDPREFX ( ..reg -- ..reg ) 0 SPT !  0 DUN !  USE?
   IF OPREFX32  DUN? NOT
    IF  OPREFX32  DUN? NOT SIZE32? NOT AND
     IF 146 C, THEN  THEN
   ELSE  OPREFX16 DUN? NOT
    IF  OPREFX16 DUN? NOT SIZE32? AND
     IF 146 C, THEN THEN THEN ;
: ADRPREFX ( ..Reg -- ..Reg ) 0 SPT !  0 DUN !
   USE?  IF APREFX32 DUN? NOT  IF  APREFX32 THEN
   ELSE APREFX16 DUN? NOT  IF  APREFX16 THEN THEN ;
: PREFX ( ..reg opadr. -- ..reg opadr)
    ADRPREFX  OPNDPREFX  SEGOVR? ;
 VARIABLE OP
: OFFPREFX ( ..op -- ..op ) OP C! OPSET OFF PREFX OP C@ ;
: ONPREFX  ( ..op -- ..op ) OP C! OPSET  ON PREFX OP C@ ;
```

SCREEN 23

```
\ OPND, OPADR;  WMEM, R/M,  WR/SM,                   10jul88 JBD

 VARIABLE OPND  VARIABLE OPADR

: WMEM,  ( disp mem reg op -- )  OVER  W,  MEM, ;

: R/M, ( mr reg -- )  OVER REG? IF  RR, ELSE MEM, THEN ;

: WR/SM,   ( rm reg op -- )   2 PICK DUP REG?
   IF  W, RR,  ELSE  DROP SIZE, MEM,  THEN  SIZE ON  ;
```

SCREEN 24

```
\ 1MI, 2MI                                           10jul88 JBD
: 1MI   CREATE  C,  DOES>  C@ C,  ;
HEX
 37 1MI AAA  3F 1MI AAS  F8 1MI CLC  FC 1MI CLD  FA 1MI CLI
 F5 1MI CMC  27 1MI DAA  2F 1MI DAS  F4 1MI HLT  CE 1MI INTO
 9F 1MI LAHF F0 1MI LOCK 90 1MI NOP  F2 1MI REP  F3 1MI REPE
 F2 1MI REPNE   F2 1MI REPNZ   F3 1MI REPZ   9E 1MI SAHF
 F9 1MI STC  FD 1MI STD  FB 1MI STI  9B 1MI WAIT D7 1MI XLAT
OCTAL

: 2MI   CREATE  C,  DOES>  C@ C,  12 C,  ;
HEX  D5 2MI AAD  D4 2MI AAM OCTAL
```

SCREEN 25

```
\ .386 etc, SHORT etc.                               10jul88 JBD

VARIABLE .386VAR    .386VAR OFF
: .386? .386VAR @ 0<> ;          ( all for 3MI use)
: .386  .386? IF  .386VAR ON ELSE .386VAR OFF THEN ; ( toggles)

VARIABLE SHORT
( Use SH before 3MI words for short jump when 386 enabled)
: SH  SHORT ON  ;
: SH? SHORT @ 0<> ;
```

SCREEN 26

```
\ 3MI, JA etc.                                       10jul88 JBD

: 3MI   CREATE  C,  DOES>  .386? NOT
   IF  C@ C,  HERE - 1-
    DUP -200 177 WITHIN NOT ABORT" Branch out of Range" C,
   ELSE ( all 386 cases) C@ OFFPREFX   SH?
    IF SHORT OFF  C,  #)?
     IF HERE - 1- C,
     ELSE ( D#)) HERE 4 + S>D  D-  DROP C, THEN
    ELSE ( 386 near, not short)  17 C, 20 + C, #)?
     IF  HERE - 2- ,
     ELSE ( D#)) HERE 4 + S>D D- SWAP , ,
   THEN THEN THEN  WRAP ;
```

SCREEN 27

```
\ 3MI  words                                         10jul88 JBD
HEX
 77 3MI JA   73 3MI JAE   72 3MI JB    76 3MI JBE   72 3MI JC
 74 3MI JE   7F 3MI JG    7D 3MI JGE   7C 3MI JL    7E 3MI JLE
 76 3MI JNA  72 3MI JNAE  73 3MI JNB   77 3MI JNBE  73 3MI JNC
 75 3MI JNE  7E 3MI JNG   7C 3MI JNGE  7D 3MI JNL   7F 3MI JNLE
 71 3MI JNO  7B 3MI JNP   79 3MI JNS   75 3MI JNZ   70 3MI JO
 7A 3MI JP   7A 3MI JPE   7B 3MI JPO   78 3MI JS    74 3MI JZ

OCTAL
```

SCREEN 28

```
\ 4MI, 14MI                                          10jul88 JBD
OCTAL
: 4MI   CREATE  C,  DOES>  C@ ONPREFX
   C,  MEM, WRAP ;
HEX  C5 4MI LDS  8D 4MI LEA  C4 4MI LES  OCTAL

( 14MI is 386 instrucs not covered by 4MI)

: 14MI   CREATE   C,  DOES>  C@ ONPREFX  17 C,
   C,  MEM, WRAP ;
HEX  B4 14MI LFS   B5 14MI LGS   B2 14MI LSS OCTAL
```

SCREEN 29

```
\ 5MI                                                10jul88 JBD
: 5MI   CREATE  C,  DOES>   ( no numeric operands)
   0 ( dummy param for PREFX) SWAP C@  OFFPREFX NIP
```

```
   SIZE,  WRAP  ;
 ( Use with BY, WD or DW to give opnd size, with optional
   seg override for source string; dest. uses auto ES: override)
 HEX  A6 5MI CMPS  A4 5MI MOVS  AE 5MI SCAS
: CMPSB  A6 C, ;
: CMPSW  WD OPSET OFF 0 PREFX DROP A7 C, WRAP ;
: CMPSD  DW OPSET OFF 0 PREFX DROP A7 C, WRAP ;
: MOVSB  A4 C, ;
: MOVSW  WD OPSET OFF 0 PREFX DROP A5 C, WRAP ;
: MOVSD  DW OPSET OFF 0 PREFX DROP A5 C, WRAP ;
: SCASB  AE C, ;
: SCASW  WD OPSET OFF 0 PREFX DROP AF C, WRAP ;
: SCASD  DW OPSET OFF 0 PREFX DROP AF C, WRAP ;     OCTAL
```

SCREEN 30

```
\ 6MI, LODS etc.; 7MI, DIV etc.                        10jul88 JBD
 ( Use with BY, WD or DW to give opnd size)
: 6MI   CREATE  C,  DOES>  C@ 0 SWAP OFFPREFX
   SWAP DROP  SIZE, WRAP ;
HEX  AC 6MI LODS  AA 6MI STOS
: LODSB ( no opnds) AC C, ;
: LODSW ( no opnds) DX  OPSET OFF PREFX  DROP  AD C, ;
: LODSD ( no opnds) EDX OPSET OFF PREFX  DROP  AD C, ;
: STOSB ( no opnds) AA C, ;
: STOSW ( no opnds) DX  OPSET OFF PREFX  DROP  AB C, ;
: STOSD ( no opnds) EDX OPSET OFF PREFX  DROP  AB C, ;  OCTAL
: 7MI   CREATE  C,  DOES> C@ OFFPREFX 366 WR/SM, WRAP ;
```

SCREEN 31

```
\ 8MI: IN, OUT;  9MI: DEC, INC                         10jul88 JBD

: 8MI   CREATE  C,  DOES>  C@ OP C! OPSET ON PREFX OP C@
   SWAP DUP R16? SWAP R32? OR 1 AND OR  SWAP # =
   IF  C, C,  ELSE  ( DX) 10 OR  C,  THEN  WRAP ;

HEX E4 8MI IN  E6 8MI OUT  OCTAL

: 9MI   CREATE  C,  DOES>  C@ OP C! OPSET OFF PREFX OP C@
   OVER DUP R16? SWAP R32? OR
    IF  100 OR SWAP RLOW OP,
    ELSE  376 WR/SM, THEN   WRAP ;

HEX  8 9MI DEC  0 9MI INC  OCTAL
```

SCREEN 32

```
\ 10MI, RCL etc.                                       10jul88 JBD
   ( 1 # m/r shl,  cl m/r shl,  imm8 # m/r shl are legal forms)

: 10MI CREATE C, DOES> C@ OP C!  OPSET ON PREFX OP C@
    SPT @ 1+ ROLL ( CL or # ) CL =
    IF  322 WR/SM,
    ELSE ( #)
     SPT @ 1+ ROLL ( imm8 data)  DUP 1 =
     IF  DROP 320  WR/SM,
     ELSE ( imm8) OPND !  300 WR/SM, OPND @ C,
    THEN THEN  WRAP ;

HEX   10 10MI RCL   18 10MI RCR    0 10MI ROL   8 10MI ROR
      38 10MI SAR   20 10MI SHL   20 10MI SAL  28 10MI SHR
OCTAL
```

SCREEN 33

```
\ 11MI, CALL and JMP                                   10jul88 JBD

: 11MI   CREATE  C, C,  DOES>  OPADR ! OPSET OFF PREFX
   OPADR @ OVER DUP OPND ! DUP #)? SWAP D#)? OR
   IF  NIP C@ INTER @
     IF  1 AND IF  352  ELSE  232  THEN  C, OPND @ #)?
      IF SWAP , , ELSE -ROT SWAP , , , THEN INTER OFF
     ELSE OPND @ #)?
      IF  SWAP HERE - 2- SWAP  2DUP 1 AND SWAP BIG? NOT AND
       IF  2 OP, C,  ELSE  C,  1- ,  THEN
      ELSE ( D#)) -ROT HERE 5 + S>D D- ROT C, SWAP , , THEN THEN
   ELSE  OVER S#) =
    IF NIP #) SWAP  ELSE OVER SD#)?
     IF NIP D#) SWAP THEN THEN
   377 C, 1+ C@ ?FAR  R/M,  THEN  WRAP ;
HEX  10 E8 11MI  CALL    20 E9 11MI JMP      OCTAL
```

SCREEN 34

```
\ 12MI, PUSH and POP                                   10jul88 JBD
: 12MI ( immed, 32segreg{2bytes}, m/r, segreg, reg opcodes -- )
        CREATE  C, C, C, C, C, C, DOES>
   OPADR ! OPSET OFF PREFX OPADR @ OVER REG?
   IF  C@ SWAP RLOW OP,
   ELSE  1+ OVER SEG16?
    IF  C@ RLOW SWAP RMID OP,
    ELSE  OVER UMEMA?
     IF  COUNT SWAP C@ C,  MEM,
     ELSE 2+ OVER SEG32?
      IF COUNT C, C@ OVER FS = IF C, ELSE 10 + C, THEN  DROP
      ELSE ( Immed: PUSH only) 2+ SWAP D#?
       IF C@   C, SWAP , ,
       ELSE ( # ) C@ ( disp op) SWAP DUP BIG?
        IF SWAP C, , ELSE ( 8 bits) SWAP 2 OR  C, C,
    THEN THEN THEN THEN THEN THEN  WRAP ;
```

SCREEN 35

```
\ 12MI, PUSH and POP opcodes                           10jul88 JBD

HEX

68 0A0 0F 0FF 36 50  12MI PUSH

0  0A1 0F 8F  07 58  12MI POP
OCTAL
```

SCREEN 36

```
\ NROLL :  TOS to N+1th stack position                 10jul88 JBD

( 1 NROLL = SWAP, 2 NROLL = -ROT )
VARIABLE NUMROLL ( number to ROLL)

: NROLL ( n --)  DUP 0<>             ( for 13MIMEM use)
   IF  DUP NUMROLL !  0 DO
    NUMROLL @  ROLL LOOP   ELSE DROP  THEN ;
```

SCREEN 37

```
\ 13MI: 13MISIMM                                       10jul88 JBD

 : 13MISIMM ( immed. source with reg dest)
    OPND @ #?
    IF   OVER B/L? OVER DUP  R16?  SWAP R32? OR 2DUP AND
     -ROT 1 AND SWAP NOT 2 AND OR 200 OP,
     SWAP RLOW 300 OR  OP @  OP,  ,/C,
    ELSE ( D# source) 177777 DUP 2DUP AND
     -ROT 1 AND SWAP NOT 2 AND OR 200 OP,
     SWAP RLOW 300 OR  OP @  OP, DROP SWAP , ,
    THEN  ;
```

SCREEN 38

```
\ 13MI: 13MIMEM                                        10jul88 JBD

: 13MIMEM  ( dest= mem cases of 13MI)
   SPT @  ROLL  DUP REG?
   IF  OP C@  WMEM,
   ELSE  ( #)  #?
    IF SPT @  PICK B/L? DUP NOT 2 AND 200 OR SIZE,
     SPT @  NROLL  OP @  MEM,
     SIZE @ AND ,/C,
    ELSE ( D#) 177777 DUP NOT 2 AND 200 OR SIZE,
     SPT @  NROLL  OP @  MEM,
     DROP SWAP , ,  THEN  THEN  ;
```

SCREEN 39

```
\ 13MI, ADD etc.                                       10jul88 JBD

: 13MI  CREATE C, C, DOES>  COUNT OP C!  C@ LOGICAL !
   OPSET ON PREFX  DUP REG? ( dest a reg?)
   IF  OVER REG?  ( source a reg also?)
    IF OP @ OVER W, SWAP RR,
    ELSE  OVER DUP UMEM? SWAP U#)?   OR   ( memory source?)
     IF  OP @ 2 OR WMEM,
     ELSE  ( # or D#) OVER OPND  !  NIP DUP RLOW 0= ( accum?)
      IF  OP @ 4 OR OVER W, OPND @ #?
       IF  R16? ,/C,  ELSE ( D#) DROP  SWAP , , THEN
      ELSE 13MISIMM ( immed. source, dest reg but not accum)
    THEN THEN THEN
   ELSE  ( mem dest.)  13MIMEM THEN  WRAP ;
```

SCREEN 40

```
\ 15MI, SETcond                                        10jul88 JBD

: 15MI  CREATE  C, DOES> C@ OFFPREFX  17 C,  220 OR C,
   DUP R8?
    IF  RLOW 300 OP,
    ELSE ( mem) 0 ( rmid) MEM, THEN  WRAP ;

HEX
7 15MI SETA 3 15MI SETAE  2 15MI SETB 6 15MI SETBE 2 15MI SETC
4 15MI SETE F 15MI SETG 0D 15MI SETGE 0C 15MI SETL 0E 15MI SETLE
 6 15MI SETNA  2 15MI SETNAE  3 15MI SETNB  7 15MI SETNBE
 3 15MI SETNC  5 15MI SETNE  0E 15MI SETNG 0C 15MI SETNGE
0D 15MI SETNL 0F 15MI SETNLE  1 15MI SETNO 0B 15MI SETNP
 9 15MI SETNS  5 15MI SETNZ   0 15MI SETO  0A 15MI SETP
0A 15MI SETPE 0B 15MI SETPO   8 15MI SETS   4 15MI SETZ
OCTAL
```

SCREEN 41

```
\ 16MI + 17MI, CBW,CWD etc, PUSHA/POPA etc, IRET/D   10jul88 JBD

: 16MI  CREATE C, DOES> USE? IF 146 C, ( 66h) THEN
   C@ C, WRAP ;
HEX  99 16MI CWD   98 16MI CBW  60 16MI PUSHA  9C 16MI PUSHF
     61 16MI POPA  9D 16MI POPF CF 16MI IRET
OCTAL

: 17MI  CREATE C, DOES> USE? NOT IF 146 C, ( 66h) THEN
   C@ C, WRAP ;

HEX  99 17MI CDQ  98 17MI CWDE  60 17MI PUSHAD  9C 17MI PUSHFD
```

```
     61 17MI POPAD  9D 17MI POPFD  CF 17MI IRETD
OCTAL
```

SCREEN 42

```
\ 18MI,  SHLD/SHRD ( non-standard modr/m byte)        10jul88 JBD
   ( cl reg m/r shld,  imm8 # reg m/r shld  are legal forms)
VARIABLE CLFLG      : CL? CL = 0<> ;     : CL  CL CLFLG ON ;
: CLFLG? CLFLG @ 0<> ;
: 18MI CREATE C, DOES> C@  ONPREFX   17 C,
   SPT @ 2+ ROLL ( CL or # )  CL?
    IF  1+ C,
    ELSE ( # ) SPT @ 2+ ROLL  OPND C!  C, THEN
   DUP REG? ( dest a reg?)
    IF ( source a reg also) SWAP RR,  CLFLG?
     IF CLFLG OFF ELSE  OPND C@ ( imm8) C, THEN
    ELSE ( dest mem, source reg)
     SPT @ ROLL MEM,  CLFLG?
     IF CLFLG OFF ELSE  OPND C@ ( imm8) C, THEN
    THEN  WRAP ;
HEX   A4 18MI SHLD  AC 18MI SHRD   OCTAL
```

SCREEN 43

```
\ 19MI, LAR + LSL, BSF + BSR                          10jul88 JBD

: 19MI CREATE C, DOES> C@  ONPREFX  17 C, C,
   OVER REG?  ( source a reg also?)
    IF   RR,
    ELSE ( mem source)  MEM, THEN  WRAP ;

HEX  02 19MI LAR   03 19MI LSL
     BC 19MI BSF   BD 19MI BSR   OCTAL
```

SCREEN 44

```
\ 20MI, LGDT etc.                                     10jul88 JBD

  ( 2nd op, rmid -- )
: 20MI  CREATE  C, C, DOES> DUP OPADR !  C@ OFFPREFX
   17 C,  C, OPADR @ 1+ C@ ( rmid)
   OVER REG?
    IF  SWAP RLOW  OR  300 OP,
    ELSE ( mem)  MEM, THEN  WRAP ;

HEX  10 1 20MI LGDT   18 1 20MI LIDT   10 0 20MI LLDT
     18 0 20MI LTR     0 1 20MI SGDT    8 1 20MI SIDT
      0 0 20MI SLDT   20 1 20MI SMSW    8 0 20MI STR
     20 0 20MI VERR   28 0 20MI VERW
OCTAL
```

SCREEN 45

```
\ 21MI, BT  etc.                                      10jul88 JBD
   ( reg m/r bt,  imm8 # m/r bt  are legal forms)
      ( N.B.: non-standard modr/m byte!)
: 21MI CREATE C, DOES> C@ OP C! OPSET ON PREFX   17 C,
   SPT @  ROLL ( reg or  # ) DUP #?  ( source immed?)
    IF  DROP 272 C, SPT @ ROLL OPND C! DUP REG? ( dest a reg?)
     IF RLOW 300 OR  OP C@ OR C,  OPND C@ C,
     ELSE ( mem dest) OP C@ MEM, OPND C@ C, THEN
    ELSE ( reg source ) OPND !  OP C@ 203 OR C,
     DUP REG? ( dest a reg also?)
      IF OPND @ ( source reg)  RR,
      ELSE ( dest mem, source reg)  OPND @ MEM, THEN
    THEN  WRAP ;

HEX  20 21MI BT   38 21MI BTC   30 21MI BTR   28 21MI BTS
OCTAL
```

SCREEN 46

```
\ 22MI, INS etc.                                      10jul88 JBD
: 22MI   CREATE  C,  DOES>   ( DX -- )
   SWAP DROP  ( DX not needed in code)
   0 ( dummy param for PREFX ) SWAP C@  OFFPREFX NIP
   SIZE,  WRAP  ;
 ( Use with BY, WD or DW to give operand size.)
HEX  6C 22MI INS   6E 22MI OUTS

: INSB   6C C, ;
: INSW   WD OPSET OFF 0 PREFX DROP  6D C, WRAP ;
: INSD   DW OPSET OFF 0 PREFX DROP  6D C, WRAP ;
: OUTSB  6E C, ;
: OUTSW  WD OPSET OFF 0 PREFX DROP  6F C, WRAP ;
: OUTSD  DW OPSET OFF 0 PREFX DROP  6F C, WRAP ;
OCTAL
```

SCREEN 47

```
\ 23MI, MOVSX and MOVZX                               10jul88 JBD

: 23MI CREATE C, DOES> C@  ONPREFX  17 C,
   2 PICK  R8? IF C, ELSE  SIZE, THEN
   OVER REG?  ( source a reg also?)
    IF  RR,
    ELSE ( mem source)  MEM, THEN WRAP ;

HEX  BE 23MI MOVSX  B6 23MI MOVZX  OCTAL
```

SCREEN 48

```
\ TEST: TESTMEM                                       10jul88 JBD

: TESTMEM  ( dest= mem cases of TEST)
   SPT @    ROLL  DUP REG?
   IF  204  WMEM,
   ELSE  ( # )  #?
    IF  366 SIZE, 0 MEM, SIZE @ ,/C,
    ELSE ( D# ) 366 SIZE, 0 MEM, SWAP  , ,
   THEN  THEN  ;
```

SCREEN 49

```
\ TEST                                                10jul88 JBD

: TEST  OPSET ON PREFX
   DUP REG? ( dest a reg?)
   IF  OVER REG?  ( source a reg also?)
    IF 204  OVER W, SWAP RR,
    ELSE  OVER DUP UMEM? SWAP U#)?   OR  ( memory source?)
     IF  204  WMEM,
     ELSE  ( # or D# ) OVER OPND  !  NIP DUP RLOW 0= ( ACC? )
      IF  250  OVER W,
      ELSE 366 OVER W, DUP RLOW 300 OP, THEN
      DUP R32? IF ( #D) DROP SWAP , ,
       ELSE R16?  ,/C,  THEN THEN THEN
   ELSE  ( mem dest.)  TESTMEM  THEN  WRAP ;
```

SCREEN 50

```
\ ESC, INT, XCHG                                      10jul88 JBD
HEX

: ESC  ( rm, 6-bit const -- ) RLOW  0D8  OP, R/M, ;

: INT  ( n -- )  0CD  C, C,  ;  ( N.B.: no # )

: XCHG   ( mr1 mr2 -- )   OPSET ON PREFX  DUP REG?
   IF  DUP DUP AX =  SWAP EAX = OR
     IF  DROP RLOW 90 OP,  ELSE  OVER DUP AX =  SWAP EAX = OR
     IF  NIP  RLOW 90 OP,  ELSE  86 WR/SM,  THEN  THEN
   ELSE  ROT 86 WR/SM,  THEN  WRAP ;
```

SCREEN 51

```
\ MOV: MOVRGSG2                                       10jul88 JBD

: MOVRGSG2 ( -- ss dst )       ( Continuation from MOVRGSG1)
    OVER SEG?
    IF  SWAP 8C C, RR,
    ELSE  OVER DUP  #?   SWAP D#?  OR
     IF   DUP DUP  R16? SWAP R32?  OR  SWAP
      RLOW OVER 8 AND OR B0 OP,
      SWAP D#?  IF  DROP SWAP , ,  ELSE  ,/C,  THEN
     ELSE  8A OVER W, R/M,  THEN THEN ;
```

SCREEN 52

```
\ MOV: MOVRGSG1                                       10jul88 JBD

: MOVRGSG1 ( -- ss dst )  ( dest either REG or SEG)
   DUP SEG?
   IF  8E C, R/M,
   ELSE  DUP REG?
    IF  ( direct memory source? )  OVER DUP
     #)?  SWAP  D#)?  OR  OVER RLOW 0= AND
      IF  A0 SWAP W,  D#)?  IF  SWAP , , ELSE  ,  THEN
      ELSE  ( all other cases )  MOVRGSG2  THEN THEN THEN ;
```

SCREEN 53

```
\ MOV: MOVMEM                                         10jul88 JBD
  ( dest a memory expression, so source is reg or immed.)

: MOVMEM   ( ss dst -- )    ( dest a memory expression)
   SPT @  ( PREFX handles increment for double displacements)
   ROLL  DUP SEG?  ( source a segreg?)
   IF  8C C, MEM,
   ELSE  DUP #?
    IF  DROP C6 SIZE, 0 MEM,  SIZE @ ,/C,
    ELSE  DUP D#?
     IF  DROP C6 SIZE, 0 MEM,  SWAP , ,
     ELSE  OVER #)?  OVER RLOW 0= AND
      IF  A2 SWAP W,  DROP  ,   ELSE  88 OVER W, R/M,
      THEN THEN THEN  THEN  ;
```

SCREEN 54

```
\ MOV, MOVSPL                                         10jul88 JBD

: MOVSPL  0F C, DUP SPL? ( dest SPL?)
   IF DUP CTL? IF 22 ELSE DUP DBG? IF 23 ELSE 26 THEN THEN
    C, RMID SWAP RLOW OR C0 OR C,
   ELSE ( source is SPL)  SPT @  PICK
    DUP CTL? IF DROP 20 ELSE  DBG? IF 21 ELSE 24 THEN THEN
    C, RLOW SWAP RMID OR C0 OR C, THEN ;

: MOV ( source dest--) OPSET ON PREFX
   DUP SPL?  SPT @ 1+ PICK SPL?  OR  ( dest or source SPL?)
    IF MOVSPL
    ELSE  DUP DUP REG? SWAP SEG?  OR ( dest reg or segreg?)
```

```
      IF MOVRGSG1  ELSE MOVMEM THEN THEN  WRAP ;
```

SCREEN 55

```
\ ARPL, CLTS, BOUND, ENTER, LEAVE                    10jul88 JBD
OCTAL
 ( r16 m/r16 ARPL)
: ARPL  ( N.B.: non-standard modr/m byte!)
        OPSET ON PREFX 143 C, DUP R16?
   IF SWAP RR, ELSE ( mem dest) SPT @ ROLL MEM, THEN WRAP ;

: CLTS  ( --) 17 C,  6 C, ;

: BOUND ( mem reg bound) OPSET ON PREFX  142 C, MEM, WRAP ;

: ENTER ( imm8  imm16  enter)
   310 C,  ,  C,  ;

: LEAVE  ( --)  311 C, ;
```

SCREEN 56

```
\ JCXZ, JECXZ                                        10jul88 JBD

: JCXZ ( adr, #} or D#}  -- )  USE?
   IF 146 C, THEN  343 C,  #)?
    IF  HERE - 2- ,
    ELSE ( D#})  HERE 4 + S>D D- SWAP , ,  THEN ;

: JECXZ ( adr, #} or D#} -- )  USE? NOT
   IF 146 C, THEN  343 C,  #)?
    IF  HERE - 2- ,
    ELSE ( D#})  HERE 4 + S>D D- SWAP , ,  THEN ;
```

SCREEN 57

```
\ 7MI and 13MI, Opcode Definitions.                  10jul88 JBD
( Put here to avoid conflicts with ordinary NOT, AND and OR)
 HEX
 30 7MI DIV  38 7MI IDIV  28 7MI IMUL 20 7MI MUL 10 7MI NOT

 0 10 13MI ADC   0  0 13MI ADD   2 20 13MI AND  0 38 13MI CMP
 2  8 13MI  OR   0 18 13MI SBB   0 28 13MI SUB  2 30 13MI XOR

DECIMAL
```

SCREEN 58

```
\ Structured Conditionals                            10jul88 JBD
: A?>MARK    ( -- f addr ) TRUE   HERE   0 C,   ;
: A?>RESOLVE ( f addr -- ) HERE OVER 1+ - SWAP C! ?CONDITION ;
: A?<MARK    ( -- f addr ) TRUE   HERE   ;
: A?<RESOLVE ( f addr -- ) HERE 1+ -  C,   ?CONDITION   ;
' A?>MARK    ASSEMBLER IS ?>MARK
' A?>RESOLVE ASSEMBLER IS ?>RESOLVE
' A?<MARK    ASSEMBLER IS ?<MARK
' A?<RESOLVE ASSEMBLER IS ?<RESOLVE
HEX
75 CONSTANT 0=   74 CONSTANT 0<>   79 CONSTANT 0<
78 CONSTANT 0>=  7D CONSTANT <     7C CONSTANT >=
7F CONSTANT <=   7E CONSTANT >     73 CONSTANT U<
72 CONSTANT U>=  77 CONSTANT U<=   76 CONSTANT U>
71 CONSTANT OV
DECIMAL
```

SCREEN 59

```
\ Structured Conditionals                            10jul88 JBD
HEX
: IF      C,   ?>MARK  ;
: THEN    ?>RESOLVE   ;
: ELSE    0EB IF   2SWAP   THEN   ;
: BEGIN   ?<MARK   ;
: UNTIL   C,   ?<RESOLVE   ;
: AGAIN   0EB UNTIL   ;
: WHILE   IF   ;
: REPEAT   2SWAP   AGAIN   THEN   ;
: DO      # CX MOV   HERE   ;
: NEXT    >NEXT #) JMP   ;
: 1PUSH   >NEXT 1- #) JMP   ;
: 2PUSH   >NEXT 2- #) JMP   ;
DECIMAL
```

End Listing

80386 PROTECTED MODE

Listing One *(Listing continued, text begins on page 36.)*

```
comment #*********************************
Program by  Neal Margulis -- Use MASM 5.0
#*****************************************

descriptor STRUC
    limit_0_15  dw  0       ;  lowest 16 bits of segment limit
    base_0_15   dw  0       ;  lowest 16 bits of base
    base_16_23  db  0       ;  base bits 16-23
    access      db  0       ;  Present bit, priv. level, type
    gran        db  0       ;  G bit, D/B bit , limit bits 16-19
    base_24_31  db  0       ;  base bits 24-31
descriptor ENDS

code_seg_access equ 09AH    ; Present, DPL=0, non-conforming,read/
                              exec
data_seg_access equ 092H    ; Present, DPL=0, Expand-Up,writeable

; have screenbase equal B8000H for EGA or B0000H for monochrome
screenbase   EQU  0B8000H
screenseg    EQU  0B800H

CSEG segment word  use16 'code'
assume  cs:CSEG,ds:CSEG

       mov ax,CSEG
       mov ds, ax

; Make entries in GDT for PMODE segment as code or data
       mov ax, seg PMODE
       and eax, 0FFFFh
       shl eax, 4H
       mov ebx, eax
       shr eax, 16
       mov gdt_PM_1.base_0_15, bx
       mov gdt_PM_2.base_0_15, bx
       mov gdt_PM_1.base_16_23,al
       mov gdt_PM_2.base_16_23,al

; Make entry in GDT for C3 segment as code
       mov ax,seg C3
       and eax, 0FFFFH
       shl eax, 4H
       mov ebx, eax
       shr eax, 16
       mov gdt_c3_5.base_0_15, bx
       mov gdt_c3_5.base_16_23,al

; Set up gdtr for lgdt instruction
       mov ax, cs
       and eax, 0FFFFH
       shl eax, 4H
       add eax, offset gdttbl
       mov dword ptr gdtaddr+2,eax
       lgdt  gdtaddr                ; set GDT address
A20_ON:
       cld                          ; Clear direction flag
       cli                          ; Disable interrupts

; Enter Protected Mode
       mov     eax,cr0
       or      eax,1
       mov     cr0,eax              ; Enable protected mode

            ;flush prefetch queue
       DB 0EAH,0H,0H,08H,0H         ; jmp to PMODE and execute

gdtaddr label   qword
       dw      48
       dd      ?
       dw      0

; global descriptor table

gdttbl   label   dword
gdt_null  descriptor <,,,,,>  ;   GDT entry 0 (null descriptor)
gdt_PM_1  descriptor <0FFFFH,,,code_seg_access,0C0H,0>; D bit ON
gdt_PM_2  descriptor <0FFFFH,,,data_seg_access,08FH,>  ;
gdt_3     descriptor <0FFFFH,0,0,data_seg_access,08FH,0>
gdt_rm_4  descriptor <0FFFFH,0,0,data_seg_access,08fH,0>
gdt_c3_5  descriptor <0FFFFH,,,code_seg_access,080H,0>; D bit OFF

CSEG ends
PMODE segment para public use32 'code'
    assume cs:PMODE
       mov ax, 18h          ;selector 18H is 4 Gigabyte data
segment with
                                                          ;
base at 0
       mov es, ax
       mov fs, ax
       mov ax, 10h          ;  Data segment with base at 'c2seg'
       mov ds, ax
       mov cx, 025h
       mov     edi,screenbase  ;  Addressing screen memory from
                               ;  protected mode
display:mov byte ptr es:[edi],'P'
       add     edi,2
       mov     byte ptr es:[edi],'M'
       add     edi,2
       mov     byte ptr es:[edi],' '
       add     edi,2
       loopne display

       db      0eah, 0h, 0h, 0h, 0h,28h, 0h ; jmp to c3 and
execute

align 16
            pdat  db 0ach
lastpm  label dword
PMODE ends

c3 segment para public use16 'code'
    assume cs:c3

       mov ax, 20h          ; Change segments back to have valid
       mov es, ax           ;  real mode attributes.
       mov ds, ax
```

```
        mov fs, ax
        mov eax,cr0
        and eax, 07fffffeh
        mov cr0,eax             ; enter real mode
        jmp far ptr flushrl     ; flush queue
flushrl:
        mov ax, screenseg       ; Address screen memory from real mode
        mov ds, ax
        sub edi, 0b8000h
        mov si,di
        mov byte ptr ds:[si],'C'    ;  Write to screen
        add si,2
        mov byte ptr ds:[si],'3'

A20_off:

        mov ah, 04ch                ;  DOS termination
        mov al, 01h
        int 21h
c3 ends
```

End Listing

```
end
```

DOUBLE CROSS

Listing One

(Text begins on page 46.)

```
 1: PROGRAM xxref; {Copyright 1987 by
    Chrysalis Software Corp.
    All rights reserved.}
 2: {Last updated 871101 :1320}
 3: {$R+}
 4: {$I-}
 5:
 6: CONST
 7:   PubSize = 10000;
 8:   RefSize = 4096;
 9:   Refshift = 12;
10:   LegalChars : Set of CHAR = ['A'..'Z'
                    ,'0'..'9','@','_'];
11:   Blanks : String[33] = '                ';
12:
13: TYPE
14:   AnyString = String[255];
15:   SomeString = String[32];
16:   StrPtrArr = ARRAY [1..PubSize] OF
                              ^SomeString;
17:   SortArray = ARRAY [Char] OF Integer;
18:   NameArray = ARRAY [1..255] OF
                             ^SomeString;
19:   FileNumArray = ARRAY [1..PubSize]
                                 OF Byte;
20:   RefPtr = Record
21:        First: Integer;
22:        Last : Integer;
23:      End;
24:   Ref =  Record
25:         Filenum : Byte;
26:         Define : Boolean;
27:         Linenum : Integer;
28:         Next : Integer;
29:       End;
30:
31:   RefPtrarr = ARRAY [1..Pubsize]
                            OF RefPtr;
32:   RefArr = ARRAY [1..Refsize] OF Ref;
33:
34:
35:
36: VAR
37:   PublicVarArr : ^StrPtrArr;
38:   PublicFileArr : FileNumArray;
39:   junk : AnyString;
40:   line : AnyString;
41:   temp : SomeString;
42:   i,j,k,l,m,n : Integer;
43:   RANum : Integer;
44:   RAOff : Integer;
45:   PublicCount,ExtrnCount : Integer;
46:   infile : text[10000];
47:   infilename : AnyString;
48:   outfile : text[10000];
49:   outfilename : AnyString;
50:   KeyLen : Integer;
51:   ArrayLength : Integer;
52:   Filename : NameArray;
53:   Filecount : Integer;
54:   name:anystring;
55:   ext:somestring;
56:   possep: Integer;
57:   Token : Somestring;
58:   TempToken : Somestring;
59:   TempRef : Ref;
60:   StartOfLine : Boolean;
61:   FoundDef : Integer;
62:   OutLine : Integer;
63:   LastSymLine : AnyString;
64:   SameSym : Boolean;
65:   SendCRs : Boolean;
66:
67:   RefPtrs : ^RefPtrArr;
68:   RefArrs : ARRAY [1..16] OF ^RefArr;
69:   RefCount : Integer;
70:   Page : Integer;
71:   Brief : Boolean;
72:
73:   Topofheap : ^Integer;
74:
75:
76: CONST
77:   Separator = '.....................
                          ..........';
78:
79:
80:
```

```
81: PROCEDURE IOcheck(FileName : Somestring);
82:
83: VAR
84:   IOCode : Integer;
85:   Ch     : Char;
86:
87: BEGIN
88:
89:   IOCode := IOResult;
90:   IF IOCode <> 0 THEN
91:     BEGIN
92:       WriteLn;
93:       CASE IOCode OF
94:         $01 : WriteLn('File ',FileName,'
                          does not exist');
95:         $02 : WriteLn('File ',FileName,'
                        not open for input');
96:         $03 : WriteLn('File ',FileName,'
                      not open for output');
97:         $04 : WriteLn('File ',FileName,' not open');
98:         $05 : WriteLn('Can''t write to ',FileName);
99:         $06 : WriteLn('Can''t read from ',FileName);
100:        $f0 : WriteLn('Disk write error on ',FileName);
101:        $f1 : WriteLn('Directory is full');
102:        $ff : WriteLn('File ',FileName,' disappeared');
103:       ELSE
104:         WriteLn('Unknown I/O error on ',FileName);
105:       END;
106:       Halt;
107:     END;
108: END;
109:
110:
111:
112:
113: FUNCTION GetDate:SomeString;
114:
115: CONST
116:   Dos_get_date = $2a;
117:   Dos_get_time = $2c;
118:
119: TYPE
120:   regpack = RECORD
121:     CASE integer OF
122:       1 : (ax,bx,cx,dx,bp,si,di,ds,es,flags:integer);
123:       2 : (al,ah,bl,bh,cl,ch,dl,dh:byte);
124:     END;
125:
126: VAR
127:   regs : regpack;
128:   result : SomeString;
129:   temp : SomeString;
130:
131: BEGIN
132:   WITH regs DO
133: BEGIN
134:       ah := dos_get_date;
135:   msdos(regs);
136:   Str(dh,result);
137:   Str(dl,temp);
138:   result := result + '/' + temp;
139:   Str(cx,temp);
140:   result := result + '/' + temp;
141:   ah := dos_get_time;
142:   msdos(regs);
143:   Str(ch,temp);
144:   IF length(temp) = 1 THEN temp := '0' + temp;
145:   result := result + ' ' + temp;
146:   Str(cl,temp);
147:   IF length(temp) = 1 THEN temp := '0' + temp;
148:   result := result + ':' + temp;
149:   Str(dh,temp);
150:   IF length(temp) = 1 THEN temp := '0' + temp;
151:   result := result + ':' + temp;
152: END;
153:
154:   GetDate := result;
155:
156: END;
157:
158:
159: PROCEDURE Megasort(VAR PtrArray:StrPtrArr; VAR SubArray1:FileNumArray;
160:                    VAR SubArray2:RefPtrArr;
161:                     KeyLength:Integer;ArraySize:Integer);
162:
163: VAR
164:   l : Char;
165:   m : Char;
166:   i : Integer;
167:   j : Integer;
168:   BucketCount  : SortArray;
169:   BucketPosition : SortArray;
170:   TempPtrArr : ^StrPtrArr;
171:   TempSubArr1: FileNumArray;
172:   TempSubArr2: ^RefPtrArr;
173:
174:
175: BEGIN
176:
177:   New(TempPtrArr);
178:   New(TempSubArr2);
179:
180:   FOR i := KeyLength DOWNTO 1 DO
181:     BEGIN
182:       FOR l := #0 TO #255 DO
183:         BucketCount[l] := 0;
184:       FOR j := 1 TO ArraySize DO
185:         BEGIN
186:           IF i > length(PtrArray[j]^) THEN
187:             m := #0
188:           ELSE
189:             m := PtrArray[j]^[i];
190:           BucketCount[m] := BucketCount[m] + 1;
191:         END;
192:
193:       BucketPosition[#0] := 1;
194:       FOR l := #1 TO #255 DO
195:         BucketPosition[l] := BucketCount[pred(l)] + BucketPosition[pred(l)];
196:
197:       FOR j := 1 TO ArraySize DO
198:         BEGIN
199:           IF i > length(PtrArray[j]^) THEN
200:             m := #0
201:           ELSE
202:             m := PtrArray[j]^[i];
203:           TempPtrArr^[BucketPosition[m]] := PtrArray[j];
204:           TempSubArr1[BucketPosition[m]] := SubArray1[j];
205:           TempSubArr2^[BucketPosition[m]] := SubArray2[j];
206:           BucketPosition[m] := BucketPosition[m] + 1;
207:         END;
208:
209:       FOR j := 1 TO ArraySize DO
210:         BEGIN
211:           PtrArray[j] := TempPtrArr^[j];
212:           SubArray1[j] := TempSubArr1[j];
```

```
213:          SubArray2[j] := TempSubArr2^[j];
214:        END;
215:
216:    END;
217:
218:
219:  END;
220:
221:
222:
223:
224:
225: PROCEDURE Canonical(VAR Old:Anystring);
                {external 'xxsubs.bin';}
226:
227: VAR
228:   i : Integer;
229:
230: BEGIN
231:
232:   IF old <> '' THEN
233:     BEGIN
234:       IF pos(';',old) > 0 THEN
235:         old := copy(old,1,pos(';',old)-1);
236:       IF old <> '' THEN
237:         BEGIN
238:           FOR i := 1 TO Length(old) DO
239:             IF old[i] = chr(9) THEN
240:               old[i] := ' '
241:             ELSE IF old[i] IN ['a'..'z'] THEN
242:               old[i] := Upcase(old[i]);
243:         END;
244:     END;
245:
246: END;
247:
248:
249:
250:
251:
252: FUNCTION Hashit(Pubs: Integer; VAR Actual:
                    Somestring):Integer;
                    {external Canonical[$52];}
253:
254: VAR
255:   i : Integer;
256:   temp : Integer;
257:
258: BEGIN
259:
260:   temp := 0;
261:
262:   IF Actual <> '' THEN
263:     BEGIN
264:       FOR i := 1 TO Length(Actual) DO
265:         temp := ((temp shl 5) + (temp shr 11) + ord
                    (Actual[i])) AND 32767;
266:       Hashit := (temp MOD Pubs) + 1;
267:     END
268:   ELSE
269:     Hashit := 0;
270:
271: END;
272:
273:
274:
275:
276: FUNCTION Locate(VAR StrArray:StrPtrArr;PublicSize:Integer;
                    VAR Value:Somestring):Integer;
                    {external Canonical[$86]
277:
278: VAR
279:   i : Integer;
280:
281: BEGIN
282:
283:   i := Hashit(Publicsize,Value);
284:   WHILE (StrArray[i] <> nil) AND (StrArray[i]^ <> Value) DO
285:     BEGIN
286:       i := i + 1;
287:       IF i > Publicsize THEN i := 1;
288:     END;
289:
290:   IF StrArray[i] = nil THEN
291:     Locate := 0
292:   ELSE
293:     Locate := i;
294:
295: END;
296:
297:
298:
299:
300: PROCEDURE RemoveToken(VAR Line:Anystring; VAR Temp:
                          Somestring);
                          {external Canonical[$100];}
301:
302: VAR
303:   i,j,k : Integer;
304:   TempChar : Char;
305:
306: BEGIN
307:
308:   Temp := '';
309:
310:   IF Line <> '' THEN
311:     BEGIN
312:       i := 1;
313:       WHILE (i <= length(line)) AND NOT(Line[i]
                  IN LegalChars) DO
314:         i := i + 1;
315:       IF i <= length(line) THEN
316:         BEGIN
317:           j := i;
318:           WHILE (j <= length(line)) AND (Line[j]
                      IN LegalChars) DO
319:             j := j + 1;
320:           Temp := Copy(Line,i,j-i);
321:           Line := Copy(Line,j,255);
322:         END
323:       ELSE
324:         BEGIN
325:           Temp := '';
326:           Line := '';
327:         END;
328:     END
329:   ELSE
330:     BEGIN
331:       Temp := '';
332:       Line := '';
333:     END;
334: END;
335:
336:
337:
338:
339: PROCEDURE NextToken(VAR Line:Anystring; VAR Temp:Somestring);
                                        {external Canonical[$1b2];}
340:
341: VAR
342:   i,j,k : Integer;
343:   TempChar : Char;
344:
345: BEGIN
346:
347:   Temp := '';
348:
349:   IF Line <> '' THEN
350:     BEGIN
351:       i := 1;
352:       WHILE (i <= length(line)) AND NOT(Line[i] IN LegalChars)
                                                          DO
353:         i := i + 1;
354:       IF i <= length(line) THEN
355:         BEGIN
356:           j := i;
357:           WHILE (j <= length(line)) AND (Line[j] IN LegalChars)
                                                          DO
358:             j := j + 1;
359:           Temp := Copy(Line,i,j-i);
360:         END
361:       ELSE
362:         BEGIN
363:           Temp := '';
364:         END;
365:     END
366:   ELSE
367:     BEGIN
368:       Temp := '';
369:     END;
370: END;
371:
372:
373:
374:
375: FUNCTION LookAhead(VAR Line:Anystring):Boolean;
376:
377: VAR
378:   i :'Integer;
379:   Temp : Boolean;
380:   Found : Boolean;
381:
382: BEGIN
383:
384:   Temp := FALSE;
385:   Found := FALSE;
386:
387:   i := 1;
388:   WHILE (i < length(line)) AND (line[i] = ' ') DO
389:     i := i + 1;
390:
391:   IF line[i] = ':' THEN
392:     BEGIN
393:       IF i = length(line) THEN
394:         BEGIN
395:           Temp := TRUE;
396:           Found := TRUE;
397:         END
398:       ELSE   {check the next char(s)}
399:         BEGIN
400:           REPEAT
401:             i := i + 1;
402:           UNTIL (i = length(line)) OR (line[i] <> ' ');
403:           IF i = length(line) THEN {all trailing blanks}
404:             BEGIN
405:               Temp := TRUE;
406:               Found := TRUE;
407:             END;
408:         END;
409:     END;
410:
411:   IF Found = FALSE THEN
412:     IF copy(line,i,3) = 'END' THEN
413:       BEGIN
414:         Temp := FALSE;
415:         Found := TRUE;
416:       END;
417:
418:   IF Found = FALSE THEN
419:     Temp := (line[i] IN ['A'..'Z',':']);
420:
421:   LookAhead := Temp;
422:
423: END;
424:
425:
426:
427:
428: FUNCTION Store(VAR StrArray:StrPtrArr;Value:
                   Somestring):Integer;
429:
430: VAR
431:   i : Integer;
432:
433: BEGIN
```

Listing One *(Listing continued, text begins on page 46.)*

```
434:
435:    i := Hashit(Pubsize,Value);
436:    WHILE (StrArray[i] <> nil) DO
437:      BEGIN
438:        i := i + 1;
439:        IF i > Pubsize THEN i := 1;
440:      END;
441:
442:    Store := i;
443:
444: END;
445:
446:
447:
448: PROCEDURE Header(Rewrite:Boolean);forward;
449:
450:
451:
452: PROCEDURE WriteOutCr;
453:
454: VAR
455:    i : Integer;
456:
457: BEGIN
458:
459:    IF SendCRs = TRUE THEN
460:      BEGIN
461:        WriteLn(outfile);
462:        IOCheck(outfilename);
463:
464:        IF Brief THEN exit;
465:
466:        OutLine := OutLine + 1;
467:        IF OutLine > 60 THEN
468:          BEGIN
469:            WriteLn(outfile);
470:            IOCheck(outfilename);
471:            WriteLn(outfile,'
                  (* = line where symbol is defined)');
472:            IOCheck(outfilename);
473:            Write(outfile,chr(12));
474:            IOCheck(outfilename);
475:            OutLine := 0;
476:            Header(SameSym);
477:          END;
478:      END;
479: END;
480:
481:
482:
483:
484: PROCEDURE WriteOut(Line:Anystring);
485:
486: VAR
487:    i : Integer;
488:
489: BEGIN
490:    Write(outfile,Line);
491:    IOCheck(outfilename);
492:    SendCRs := TRUE;
493: END;
494:
495:
496:
497:
498: PROCEDURE WriteOutLn(Line:Anystring);
499:
500: VAR
501:    i : Integer;
502:
503: BEGIN
504:
505:    SendCRs := TRUE;
506:    WriteLn(outfile,Line);
507:    IOCheck(outfilename);
508:
509:    IF Brief THEN exit;
510:
511:    OutLine := OutLine + 1;
512:    IF OutLine > 60 THEN
513:      BEGIN
514:        WriteLn(outfile);
515:        IOCheck(outfilename);
516:        WriteLn(outfile,'
              (* = line where symbol is defined)');
517:        IOCheck(outfilename);
518:        Write(outfile,chr(12));
519:        IOCheck(outfilename);
520:        OutLine := 0;
521:        Header(SameSym);
522:      END;
523: END;
524:
525:
526:
527:
528: PROCEDURE WriteOutSym(Line:Anystring);
529:
530: VAR
531:    i : Integer;
532:
533: BEGIN
534:
535:    IF Brief THEN
536:      BEGIN
537:        WriteLn(outfile,Line);
538:        IOCheck(outfilename);
539:        SendCRs := TRUE;
540:        exit;
541:      END;
542:
543:    IF OutLine > 58 THEN
544:      BEGIN
545:        FOR i := OutLine + 1 TO 61 DO
546:          WriteLn(outfile);
547:        WriteLn(outfile);
548:        IOCheck(outfilename);
549:        WriteLn(outfile,'
              (* = line where symbol is defined)');
550:        IOCheck(outfilename);
551:        Write(outfile,chr(12));
552:        IOCheck(outfilename);
553:        OutLine := 0;
554:        Header(FALSE);
555:      END;
556:    LastSymLine := Line;
557:    WriteLn(outfile,Line);
558:    IOCheck(outfilename);
559:    SendCRs := TRUE;
560:    OutLine := OutLine + 1;
561: END;
562:
563:
564:
565:
566: PROCEDURE Header;
567:
568: VAR
569:    i : Integer;
570:    Temp : Anystring;
571:
572: BEGIN
573:
574:    OutLine := 0;
575:
576:    Str(Page,Temp);
577:    Page := Page + 1;
578:
579:    WriteOutCr;
580:    WriteOutCr;
581:    WriteOutLn('       XXREF V1.0 -
       Copyright 1987 by Chrysalis Software Corp.
              Page '+Temp);
582:    WriteOutCr;
583:    WriteOutLn('        Cross reference as of
                '+GetDate+' among files: ');
584:    WriteOutCr;
585:
586:    FOR i := 1 TO FileCount DO
587:      BEGIN
588:        WriteOut(copy(blanks,1,13-length(FileName[i]^))
                                              +FileName[i]^);
589:        IF i MOD 6 = 0 THEN
590:          WriteOutCr;
591:      END;
592:    IF FileCount MOD 6 <> 0 THEN
593:      WriteOutCr;
594:    WriteOutCr;
595:    WriteOutCr;
596:
597:    WriteOutLn('Public symbol
                                  Declared in file');
598:    WriteOutLn('-------------
                                  ----------------');
599:    WriteOutCr;
600:
601:    IF Rewrite THEN
602:      BEGIN
603:        LastSymLine := copy(LastSymLine,1,33)+
                                  '(Continued)';
604:        WriteOutLn(LastSymLine);
605:        WriteOutCr;
606:      END;
607:
608:    SendCRs := FALSE;
609:
610: END;
611:
612:
613:
614: PROCEDURE Initialize;
615:
616: BEGIN
617:
618:    Mark(Topofheap);
619:
620:    New(PublicVarArr);
621:    New(RefPtrs);
622:
623:    IF ParamCount = 0 THEN
624:      BEGIN
625:        WriteLn('The parameters are
                  (in this order):');
626:        WriteLn('Input file name (which contains
              the names of the files to process)');
627:        WriteLn('Output file name, where the
              report will be written');
628:        WriteLn('Optional switch: -b (produce
              brief output, no page headers)');
629:        Halt;
630:      END;
631:
632:    IF ParamCount = 1 THEN
633:      BEGIN
634:        infilename := ParamStr(1);
635:        Write('Output file name: ');
636:        ReadLn(outfilename);
637:        Write('Brief output? (Y/N): ');
638:        ReadLn(Temp);
639:        Temp := UpCase(Temp);
640:        Brief := Temp = 'Y';
641:      END
```

(continued on page 99)

Listing One *(Listing continued, text begins on page 46.)*

```
642:    ELSE IF ParamCount = 2 THEN
643:      BEGIN
644:        infilename := ParamStr(1);
645:        outfilename := ParamStr(2);
646:        Brief := FALSE;
647:      END
648:    ELSE IF ParamCount = 3 THEN
649:      BEGIN
650:        infilename := ParamStr(1);
651:        outfilename := ParamStr(2);
652:        Temp := ParamStr(3);
653:        Temp := UpCase(Temp[2]);
654:        Brief := Temp = 'B';
655:      END;
656:
657:    Assign(infile,infilename);
658:    IOCheck(infilename);
659:    Reset(infile);
660:    IOCheck(infilename);
661:    Assign(outfile,outfilename);
662:    IOCheck(outfilename);
663:    Rewrite(outfile);
664:    IOCheck(outfilename);
665:
666:    WriteLn;
667:    WriteLn('Reading file names.');
668:
669:    i := 0;
670:    WHILE NOT EOF(infile) DO
671:      BEGIN
672:        readln(infile,line);
673:        IOCheck(infilename);
674:        IF (line <> '') AND (line[1] <> ' ')
            AND (line[1] <> '.')
675:        AND (copy(line,13,9) <> '         0') THEN
676:          BEGIN
677:            i := i + 1;
678:            name:=copy(line,1,8);
679:            ext:=copy(line,10,3);
680:            if pos(' ',name)>0 then name:=
              copy(name,1,pos(' ',name)-1);
681:            name := name+'.'+ext;
682:            GetMem(FileName[i],length(name)+1);
683:            Filename[i]^ := name;
684:          END;
685:      END;
686:
687:    Close(infile);
688:    IOCheck(infilename);
689:
690:    Filecount := i;
691:
692:    PublicCount := 0;
693:
694:    FOR i := 1 TO Pubsize DO
695:      BEGIN
696:        PublicVarArr^[i] := nil;
697:      END;
698:
699: END;
700:
701:
702:
703: PROCEDURE Pass1;
704:
705: BEGIN
706:
707:    WriteLn;
708:    WriteLn('Pass 1');
709:    WriteLn;
710:
711:    WriteLn('Reading files: ');
712:    FOR i := 1 TO Filecount DO
713:      BEGIN
714:        Write(Filename[i]^:13);
715:        IF i MOD 6 = 0 THEN WriteLn;
716:        Assign(infile,Filename[i]^);
717:        IOCheck(FileName[i]^);
718:        Reset(infile);
719:        IOCheck(FileName[i]^);
720:        WHILE NOT EOF(infile) DO
721:          BEGIN
722:            ReadLn(infile,junk);
723:            IOCheck(FileName[i]^);
724:            Canonical(junk);
725:            NextToken(junk,temp);
726:            IF temp = 'PUBLIC' THEN
727:              BEGIN
728:                RemoveToken(junk,temp);   {get rid of "PUBLIC"};
729:                REPEAT
730:                  RemoveToken(junk,temp);
731:                  IF temp <> '' THEN
732:                    BEGIN
733:                      j := Locate(PublicVarArr^,
                            Pubsize,temp);
734:                      IF j = 0 THEN
735:                        BEGIN
736:                          j := Store(PublicVarArr^,
                              temp);
737:                          GetMem(PublicVarArr^[j],
                                length(temp)+1);
738:                          PublicVarArr^[j]^ := temp;
739:                          PublicFileArr[j] := i;
740:                          PublicCount := PublicCount + 1;
741:                        END
742:                      ELSE
743:                        j := j; {for debugging}
744:                    END;
745:                UNTIL junk = '';
746:              END;
747:          END;
748:          Close(infile);
749:          IOCheck(FileName[i]^);
750:      END;
751:
752:    WriteLn;
753:    WriteLn;
754:
755: END;
756:
757:
758:
759: PROCEDURE Pass2;
760:
761: BEGIN
762:
763:    WriteLn('Pass 2');
764:    WriteLn;
765:
766:    FOR i := 1 TO Pubsize DO
767:      RefPtrs^[i].First := 0;
768:
769:    RefCount := 0;
770:
771:    WriteLn('Reading files: ');
772:    FOR i := 1 TO Filecount DO
773:      BEGIN
774:        k := 0;
775:        Write(Filename[i]^:13);
776:        IF i MOD 6 = 0 THEN WriteLn;
777:        Assign(infile,Filename[i]^);
778:        IOCheck(FileName[i]^);
779:        Reset(infile);
780:        IOCheck(FileName[i]^);
781:        WHILE NOT EOF(infile) DO
782:          BEGIN
783:            ReadLn(infile,junk);
784:            IOCheck(FileName[i]^);
785:            k := k + 1;
786:            StartOfLine := TRUE; {we are at the
                                  beginning of the line}
787:            Canonical(junk);
788:            RemoveToken(junk,token);
789:            IF token <> 'PUBLIC' THEN
790:              BEGIN
791:                WHILE token <> '' DO
792:                  BEGIN
793:                    j := Locate(PublicVarArr^,Pubsize,Token);
794:                    IF j <> 0 THEN
795:                      BEGIN
796:                        RefCount := RefCount + 1;
797:                        IF RefCount MOD Refsize = 1 THEN
798:                          New(RefArrs[RefCount DIV RefSize+1]);
799:                        IF RefPtrs^[j].First = 0 THEN
800:                          RefPtrs^[j].First := RefCount
801:                        ELSE
802:                          BEGIN
803:                            l := RefPtrs^[j].Last;
804:                            RANum := ((l-1) Shr Refshift) + 1;
805:                            RAOff := ((l-1) AND (Refsize-1)) + 1;
806:                            RefArrs[RANum]^[RAOff].Next :=
                                                  RefCount;
807:                          END;
808:                        RefPtrs^[j].Last := RefCount;
809:                        TempRef.FileNum := i;
810:                        TempRef.LineNum := k;
811:                        TempRef.Next := 0;
812:                        TempRef.Define := FALSE;
813:                        IF StartOfLine THEN
                            {if at start of line,
                             check for def}
814:                          TempRef.Define := LookAhead(junk);
815:                        RANum := ((RefCount-1) Shr Refshift) + 1;
816:                        RAOff := ((RefCount-1) AND (Refsize-1))
                                                          + 1;
817:                        RefArrs[RANum]^[RAOff] := TempRef;
818:                      END;
819:                    StartOfLine := FALSE;
820:                    RemoveToken(junk,token);
821:                  END;
822:              END;
823:          END;
824:          Close(infile);
825:          IOCheck(FileName[i]^);
826:      END;
827:
828: END;
829:
830:
831:
832: PROCEDURE CompactResults;
833:
834: BEGIN
835:
836:    j := 1;
837:    k := 1;
838:    FOR i := 1 TO Pubsize DO
839:      BEGIN
840:        IF PublicVarArr^[i] <> nil THEN
841:          BEGIN
842:            PublicVarArr^[j] := PublicVarArr^[i];
843:            PublicFileArr[j] := PublicFileArr[i];
844:            RefPtrs^[j] := RefPtrs^[i];
845:            IF i > j THEN
846:              BEGIN
847:                PublicVarArr^[i] := nil;
848:                PublicFileArr[i] := 0;
849:                Refptrs^[i].First := 0;
```

(continued on page 148)

Huge Arrays Revisited

by Kent Porter

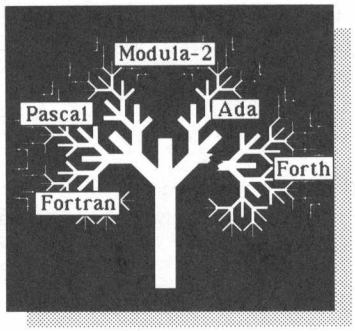
Back in March, I ran an installment of this column dealing with huge arrays: matrices that exceed the limit of the IBM PC's 64K memory segments. March is now quite a while ago, but I'm still getting letters about that column. It's generated more commentary — all positive and some pleading for help — than everything else I've done here combined. Clearly, it's a topic of great concern and more needs to be said about it. So here goes.

If you missed the March column, I'll briefly summarize the problem and the solution presented there.

An array is a collection of similar data items arranged such that they have a common identifier. Individual elements are accessed by using a *subscript*; for example, *Arr[n]* refers to the *n*th element of an array called *Arr*. The compiler inserts code that does the arithmetic necessary to convert the subscript into an offset from the start of the array, thus locating the subscripted element.

The problem is that, in the 80x86 architecture, the largest possible offset from an address is 65,535 bytes, or 64K, which is one memory segment. This means that the maximum number of elements in any given array is 64K DIV size (element). Thus an array consisting of 16-bit integers is constrained to a maximum of 32,768 elements, and an array of 8-byte real numbers can hold up to 8,192 items. That's plenty for most applications, but it rules out the PC for many scientific and engineering calculations — unless you can find a way around the segmentation barrier.

That's what the March column did. The technique presented there is to treat each row of a two-dimensional matrix as a separate array — conceptually, as a horizontal list in which each element maps to a matrix "column." This scheme allows each row of the matrix to be as large as 64K. The glue that fastens this collection of rows together is a separate "vertical" array of pointers,



each indicating a row. You can then follow a pointer into a row and access an individual column with Pascal notation such as *P^[r].col[n]* which indicates the *n*th column of the *r*th row.

This is ugly notation, but it rather elegantly solves an even uglier problem. It's not the only solution, though, so let's discuss a couple of others.

The Basic Solution

Before you laugh too hard at the notion of using Basic, you'd better look at what's happened to the language lately. A lot of folks still use Basic, which means there is a demand, which in turn spawns innovation. The new breed of Basic compilers have abolished old bugaboos such as line numbers and spaghetti logic, replacing those nasty *GOTO*s with constructs that encourage disciplined, structured programming techniques. A well-written Basic program these days looks a lot like Pascal, and in most ways it's just as powerful. And (here is the punch line) Microsoft's QuickBasic 4.0 directly supports arrays greater than 64K.

This feature alone makes QuickBasic worthy of serious consideration for giant number-crunching problems. More than one computer-hacking PhD in physics has told me he has switched to QuickBasic because of its limitless no-hassle arrays. To this add a congenial integrated environment, support of user-defined aggregate data types, multiline *IF*s, linkable libraries, call-by-name subroutines, and structured *GOTO*less code, and Basic becomes a real programming language.

Listing One, on page 136, shows HUGEMATS.BAS, written with QuickBasic. This program is a direct translation of the Pascal program that appeared in the March column (pages 86–88). Because no special routines or structures are required to implement huge arrays in QuickBasic, the Basic version is half the length of the Pascal original: 58 lines vs. 108.

The *$DYNAMIC* metacommand in the program heading tells the QuickBasic compiler to use the heap for the arrays, rather than allocating static memory for them. This is required when using huge arrays. So is the */AH* switch on the compiler command line. These are the only extras needed. Note that, in all other ways, working with huge arrays is identical to working with normal arrays.

I should point out, though, that the syntactic convenience of huge arrays in QuickBasic has a penalty. The compiler is obviously doing some stuff with mirrors and smoke in order to manage arrays limited only by available memory. This hidden trickery shows up in decreased performance. The QuickBasic version of the program runs on my 8-MHz AT clone in 25.66 seconds; it takes Turbo Pascal, Version 4.0, 5.49 seconds to do exactly the same work. This is a 4.7-to-1 performance ratio, which is pretty poor. The scales tip the other way on .EXE size, though less dramatically: 3,965 bytes for Basic- and 4,992 for Pascal. Some of this difference is explained by the fact that QuickBasic doesn't build a Ctrl-Break handler into the .EXE as Turbo Pascal does.

So the message is that if you've got a lot of huge-array crunching to do, QuickBasic offers an easier way to do it than my method in Pascal, but you pay a heavy price in performance for that convenience.

The Disk-Based Array Solution

The Pascal and Basic solutions are both limited by one insuperable barrier: the amount of memory available to the program. Free memory is almost infinitely variable, being proscribed by the physical amount of

RAM, the demands of resident and running software and the DOS version, the heap range requested in the .EXE header, the size of the stack, and other factors. Therefore it's impossible to know in advance if the target machine will have enough memory to accommodate even one large array, let alone several.

I hate even to bring it up because the performance penalty is horrendous, but the one way of being reasonably certain of having enough space for huge arrays is to put them on a hard disk. A large disk with a lot of unallocated space can give you the ability to manipulate truly enormous arrays. For example, 30 Mbytes of free disk space will accommodate 3.9 million real (8-byte) numbers, or three arrays of 1,144 × 1,145 elements each. The same space can contain 15.7 million integers in three arrays each of 2,289 × 2,290 elements. We're not encroaching on Cray territory yet (and certainly not in throughput), but we're getting warm.

The secret lies in devising an algorithm that maps virtual subscripts into random-access disk records. This is fairly easy to do. The first step is to ensure that the array's data elements are stored to disk in columnwise order. That is, go left to right across the first row, then across the second row, and so on. Let's use a simple 3 × 3 matrix as an example:

```
R0  R1  R2
R3  R4  R5
R6  R7  R8
```

where *R0* becomes disk record 0, *R1* is record 1, and so forth. The disk file thus becomes a one-dimensional list of the nine elements *R0*..*R8*. If the data type of the array is *REAL*, you can declare the file in Pascal as:

```
VAR ArrayFile : FILE OF REAL;
```

You can achieve random access to these elements via virtual two-dimensional subscripts, where *R0* has the subscripts *0* and *0*, *R3* is at *1* and *0*, and so on. The conversion expression is:

```
(RowNbr * NbrOfCols) + ColNbr
```

This yields a logical record number that you can use with Turbo Pascal's *Seek* procedure to position the disk's read/write head on the desired element. For example, you might declare:

```
VAR LogRec : LONGINT;
```

and then, somewhere in the program, locate an array element with the statements:

```
LogRec := (row * 3) + col;
Seek (ArrayFile, LogRec);
```

You can then read the element into a variable of type *REAL*. If that variable is the value returned by a function, you can use the function directly in an expression, and it will look much like a normal subscripted variable.

As an example, say you define such a function to read a value from array file A as:

```
FUNCTION A (row, col : WORD)
    : REAL;
```

and a similar function to read array file B. You can then write statements such as:

```
Sum := A (r, c) + B (r, c);
```

This is only slightly different from the pure array-handling expression:

```
Sum := A [r, c] + B [r, c];
```

and it definitely furthers the cause of readability.

(Note that Modula-2 and C don't have typed files as Pascal does, so the offset used in seek operations is not a submultiple of the file's data type but instead is a byte offset. You can, however, effect logical record numbers through multiplication or division by the size of the data type.)

Although, this approach is workable and opens the doors to arrays of virtually unlimited size, it has a serious drawback. Doing a disk seek for every array variable creates an unacceptable amount of head movement. Array-intensive programs that use it spend most of their time waiting for the disk. DISKARR.PAS in Listing Two, page 136, originally accessed each disk-based array element individually; adding two 500 × 500 arrays of *REAL* took 28 minutes. After I reworked it to its present form using a row-buffering algorithm, the program ran in 5.55 minutes. Reducing head movement thus achieved a 5-to-1 performance improvement.

The row-buffering algorithm derives from the observation that most array operations proceed in columnwise order. That is, you keep accessing successive columns in the same row until you've reached the rightmost, and then you move to the next row and repeat. A secondary observation is that it's more efficient to read/write a large block of disk data than small amounts.

Consequently, you can use a variant of disk caching to store, in memory arrays, the rows of the disk-based arrays you're currently working on. In DISKARR (Listing Two), the disk arrays are 500 × 500 matrices of type *REAL*. Thus the buffer for the current row from any matrix is defined as:

```
TYPE ArrayRow = ARRAY [0 . . 499]
      OF REAL;
```

and you have three variables of type *ArrayRow*, one for each matrix. The array files themselves are defined as:

```
TYPE RowFile = FILE OF ArrayRow;
```

The logical record number for any given *ArrayRow* is therefore numerically the same as the row number, which simplifies file I/O management:

```
Seek (ArrayFile, LONGINT (row) );
```

followed by a *read* or *write* as appropriate.

Further simplifying row buffer management is a buffer control block, defined as:

```
TYPE BuffCtlBlock = RECORD
DCurrentRow : WORD;
DIsModified : BOOLEAN;
END;
```

A control block exists for each buffer. The *CurrentRow* field contains the row number of the record presently occupying the buffer. The *IsModified* field is normally *FALSE* but becomes *TRUE* if you write new data into the buffer.

A routine that furnishes or accepts disk-based array values can compare

the parametric row with the *CurrentRow* field. If different, it knows that it must fetch the new row from the disk-based array. But before doing so, it can check the *IsModified* field to find out if the current row has been changed; if *TRUE*, it must save the current row in the disk file's corresponding logical record before fetching the new row. On the other hand, if the parametric row and *CurrentRow* are the same, no disk activity occurs.

In Listing Two, functions *A*, *B*, and *C*, and also the *WriteToC* procedure, all check and update the associated buffer control block as required. The control blocks are a slight overkill within the context of this particular program, but the combination of function *C* and procedure *WriteToC* illustrates their application in more complex programs using disk-based arrays.

The elegance of the row-buffering algorithm as implemented by subroutines is illustrated by the main work-performing loop of the program. It adds the two arrays *A* and *B* to form the output matrix *C* with:

```
FOR row := 0 TO maxRow DO BEGIN
...
   FOR col := 0 TO maxCol DO
     WriteToC (row, col,
       (A (row, col) +
       DB (row, col) ) );
     END;
```

The calls to *A*, *B*, and *WriteToC* all manage row changing as appropriate.

Special functions not shown in Listing Two are necessary when an array operation doesn't follow the usual columnwise order. A case in point is matrix multiplication, which proceeds rowwise in the multiplicand and columnwise in the multiplier. For that you need a routine capable of fetching the multiplicand's row values and loading them into a buffer. Because matrix multiplication requires that there be the same number of rows in the multiplicand as there are columns in the multiplier, you can use a variable of type *ArrayRow* for the row values within the given column. A local variable of type *ArrayRow* can serve as the file input buffer, and from there you can pluck the target column values and stuff them into successive elements of the row-wise array. Having done that, you simply summarize the products of successive rowwise and columnwise multiplications and do a *WriteToC* with the final series product. It sounds complicated, but it's really quite straighforward. And it allows you to multiply matrices of virtually unlimited size, subject to the usual rules of matrix multiplication.

The DISKARR program in Listing Two creates test files for two-dimensional matrix addition and leaves them on the disk as files ARRAY.A, ARRAY.B, and ARRAY.C. If you run the program a second time and the input files (A and B) are still the same size, the program assumes that you want to use them again, so it bypasses the file-creation step. This saves execution time. Each file occupies 1,500,000 bytes, for a total of some 4.3 Mbytes. Make sure you erase these files, lest you clutter your hard disk with useless test data.

The general approach to disk-based arrays can also be applied to EMS-based arrays without a great deal of reworking. EMS is faster, but unless you're selling turnkey systems, it's safer to assume that a user has a hard disk than to assume that he or she has enough EMS to handle truly enormous arrays.

Conclusion

Programmers of systems such as the 680x0-based Mac II and SPARC-based Sun workstations don't have to inveigle their way around the arbitrary 64K segments foisted on us by IBM and Intel. For arrays that might conceivably fit within a reasonably large memory array on the PC, QuickBasic or the method suggested in the March column furnish a workable solution.

But DRAM chips are currently in short supply and command a high price, and the PC and DOS in general comprise a limited platform from the standpoint of memory. For very large arrays, the disk-based method is probably the best solution. It doesn't furnish dazzling throughput, but it will do the job. The only practical limit is the amount of free disk space.

Huge arrays are clearly a topic of great concern to the *DDJ* readership. If you have other solutions or issues, let me hear about them.

Availability

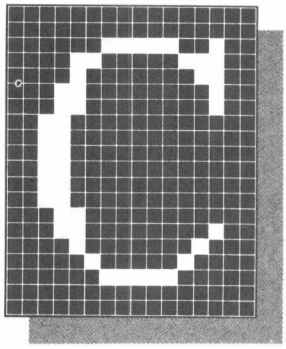
All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 136.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Screen Control, Programming as Art (?), and C++



by Al Stevens

Last month we began a running "C Column" programming project with a general-purpose video window library for the IBM PC and compatible computers. This issue continues that project with a set of window-oriented software tools using the window functions from last month. So far we haven't made the program do anything; that comes later. The final program will be a communications service driver to be used by subscribers to on-line services.

This month we add a window menu facility and a data entry screen driver. These tools are general enough to use in other projects while you wait for the communications application for which they are intended. The tools are written in C and were compiled with Turbo C, Version 1.5.

A Window Menu

The program that we are building uses a sliding bar menu at the top of the screen with a number of selections displayed horizontally across the bar. A selection can have an associated pop-down menu. To build this capability, we will use a general-purpose menu function driven by tables of structures. The structures are defined outside of the menu functions and describe the menu displays and software that each menu selection executes. Later tools, such as the text editor, will employ these menus. So will the application communications program.

Listing One, page 128, is menu.h. This file will be included in programs that must describe and execute menus by using the menu functions. You describe a menu by declaring and initializing an array of the structures that are named by the *typedef* MENU. Each element in the array describes a selection on the sliding bar menu at the top of the screen. The members of the structure describe the selection. Here is a description of each member:

mname: A pointer to the selection's name as it will be displayed in the sliding bar.

mhlpmsg: A pointer to a longer string that will be displayed on the bottom line of the screen when the selection is highlighted. This string is used to amplify the meaning of the selection by describing it to the user.

mselcs: A pointer to an array of character pointers, each of which points to the text of a selection on the pop-down menu associated with the sliding bar menu. This array must be terminated with a NULL pointer. If a NULL pointer is in the first position, no pop-down menu will appear, and the siding bar menu selection itself will execute a single command.

mshelp: An array of character pointers, each of which points to a help window mnemonic for the associated pop-down menu selection. These will be explained later when we install context-sensitive help windows.

mskeys: A pointer to a character array that contains keystrokes that will execute the pop-down commands. This feature allows a menu to be executed in one of two ways — the positioning of the menu cursor on the selection and the press of the Enter key, or the press of a designated selection key as specified in the *mskeys* array.

func: An array of function pointers that point to the functions that will be executed by the associated pop-down menu selections.

lastvsel: Used by the menu manager to remember the most recent vertical selection; should be initialized to a zero value.

With an array of MENU structures thus initialized, you call the *menu_select* function, passing it the address of the structure and an integer that says which of the horizontal selections should be highlighted when the menu is first displayed. Usually the integer has the value 1.

Listing Two, page 128, is menu.c, which contains the library functions that display the menus, get the user's selection, and execute the appropriate functions.

Listing Three, on page 129, is testmenu.c, an example of a simple menu program that shows how the menu software can be used. The example is an abstract of the menu that will be used from within the text editor part of the communications software package. The example doesn't do anything. It just illustrates how to use the menu functions. Compile and link it with the menu and window functions and it will display and navigate the menus. Figure 1, page 111, shows the menus the way they are first displayed. The arrow keys select the pop-down menus and move the cursor bars. The Enter key selects a menu item. The options menu shows an example of how you can use these menus to turn operating modes (toggles) on and off.

Compile and link testmenu.c with menu.c in Listing Two and window.c from last month's C Column. You will need Listing One, menu.h, and window.h from last month.

You can build nested layers of these menus by calling *menu_select* from within a function that was executed from a higher-level menu. In

C PROGRAMMING
(continued from page 108)

fact, the editor menu we've simulated in this example will be called from within an editor function that is called from a shell menu.

The functions that are called by the menu manager will receive two integers as parameters. These integers are the horizontal selection from the sliding menu bar (1, 2, . . , n) and the vertical selection from the pop-down menu (1, 2, . . , n). These values allow a function to determine which menu selection executed it. Each function executed from a menu must return an integer true or false value. If a false value is returned, the menu manager retains control at the selection where the function was called. If a true value is returned, the menu manager returns to the caller of the *menu_select* function. Although the menu selections in the example don't do anything, you can see this feature work in the Quit selection on the File menu. Its function returns a true value, and the menu manager returns. The other functions return false, and the user remains at the menu that executed the function.

Data Entry Screens

Listing Four, page 130, is entry.h and Listing Five, on page 130, is entry.c. These two files use the window library to implement a general-purpose data entry screen. Here's how it works: You establish a window and build an array of data entry field definitions. You write some prompting information into the window and call the data entry software, passing it the address of the field definition array. The data entry software takes over and collects the user's key entries into your buffers. All the good stuff about jumping around from field to field and help windows and such is managed by the data entry library.

The structure in entry.h named FIELD is used to describe data entry fields. You build an array of these structures terminated by a zero value structure. Here is an explanation of each of the members in the structure:

frow: The row number where the field will be displayed in the current window.

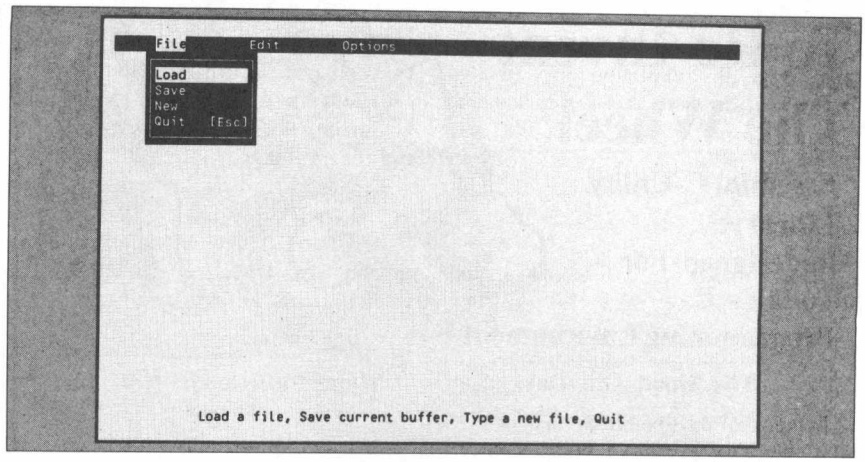


Figure 1: *Test menu (from testmenu.c)*

C PROGRAMMING
(continued from page 111)

fcol: The column number of the field. (Note: Rows and columns are relative to one. The upper-leftmost position in a window is row 1, column 1.)

fx: A running column number used by the data-entry software. This value should be initialized to 1.

fbuff: A pointer to the buffer where the data value will be stored when the user keys it in.

fmask: A pointer to a field mask. Masks are strings of underlines and other characters. The underlines correspond to field data character positions. The other characters are displayed with the field to make it more readable. For example, the mask for a telephone number might be as follows: "(_____) _____-_____"

fhelp: A pointer to the field help window mnemonic, to be explained another time. For now make it NULL.

Listing Six, on page 134 is testentr.c, XXa program that illustrates the use of the data entry screens. It establishes a window with a title and writes some prompting messages into the window. Then it calls the *data_entry* function to collect data values into its buffer. The parameters are the address of the array of FIELD structures, a TRUE value to tell the function to initialize the buffers to null-terminated strings of spaces, and the integer value 1 to tell the function to start with the field cursor on the first field in the array.

Figure 2, page 113, shows the screen that is displayed by the example after some data values have been keyed. To use the screen (which does nothing except collect data entry into a buffer) you simply type in some data values. The Ins key toggles the insert/overwrite mode. The arrow keys move the cursor. Ctrl-right or left arrows skip over words. Home goes to the beginning of a field and End goes to the end. Backspace deletes a character to the left, and Del deletes the character under the cursor. Any of the function keys or the Esc key will terminate the data entry and return the terminating key value to the caller of the *data_entry* function.

Compile and link testentr.c with entry.c in Listing Five and window.c from last month's C Column. You will need Listing Four, entry.h, and

window.h from last month.

You can keep testmenu.c and testentr.c as examples and to test other menus and data entry screens that you might design. They will not, however, be a part of the C Column programming project other than as examples. The other source files are keepers.

Next month we will add the window text editor and the context-sensitive help window library. The menu and data entry definition structures include members that point to help window mnemonics. So far, we have not used them. Later we will put help window names (string pointers) in these members and help window text into a file, and the help function will become automatic. Press the help key while the cursor is on a menu selection or a data entry field and a context-sensitive help window will pop up.

Crotchet Number 5: The Great Debate

I will now remind you that I use this column to vent my opinions on matters that are vexing. These issues are called crotchets, and they usually relate to C, although this month's crotchet spans the practice of programming without regard to any particular language.

Is programming an art or a science? In "Programming Paradigms," (*DDJ*, June 1988) Michael Swaine quotes Chuck Moore who said, "Programming is an art; we might hope it becomes a craft; it will never be a science." Later Michael dismisses programming as an art and calls it a discipline. This debate has been getting knocked around for years and I have always kept my mouth shut about it. No more — here's my nickel's worth for this month's crotchet.

Is programming an art? Webster's stresses the human side of art, as-

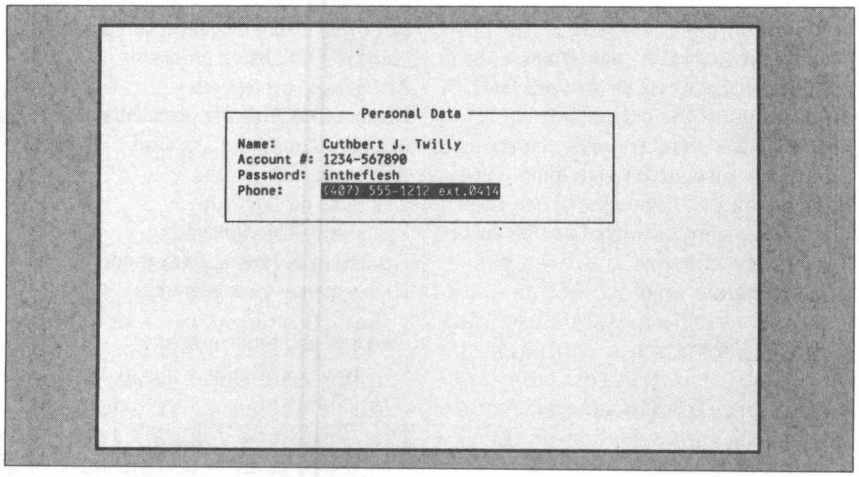


Figure 2: *A test data entry screen (from testentr.c)*

C PROGRAMMING
(continued from page 113)

signing its first definition to the "skill in performance acquired by experience, study, or observation." Note that the activity itself is not so much art as is the skill required for the practice of the activity. The next definition deals with branches of learning such as liberal and medical arts. Next comes knowledge, and then comes the creative production of aesthetic objects. Only in this last definition does the creative activity itself take part.

A computer program is the product of creative activity, but is it an aesthetic object? That depends, I suppose, on the audience. To me a work of art is something of beauty created by a skilled person to be enjoyed by the rest of us, skilled and unskilled alike. It is meant to be appreciated by the masses and is not just for kindred artists. Programs are rarely read just for the beauty of their expression and then only by other skilled programmers. Accept this notion and you must conclude that the audience for program appreciation is just too small to qualify most programs as aesthetic objects.

Is programming a science? Are programmers scientists? Scientists wear smocks, work in laboratories, and do research. They search for answers to questions and use their skills in a systematic quest for truth. They go to school for a long time to learn how to do that. Don't they? We programmers don't do that.

Is programming a discipline? Not the way I do it.

Opinions follow.

The discoveries of new formulae for concrete and paint are sciences. Architecture and portraiture are arts. Carpentry and house painting are crafts. Girder walking is a discipline.

Discovery of a new sort algorithm is science. Its publication is art. Its use is craft. Its design methodologies are disciplines, always discarded when deadlines draw near.

Perhaps many people who are not artistically gifted wish they were and thus call what they do art. My doctor does that, but there is a scar on my stomach above where my gall bladder used to be that defies aesthetic appreciation. Likewise, we call our-

selves scientists to elevate our importance several levels. If, however, we allow ourselves to be classed with house painters and drywall hangers, we tend to think we've lost caste and given up the mystique that accrues to special people. Not so.

So now you know. Programmers are craftsmen and good ol' boys (and girls, too). Pass me another Pabst Blue Ribbon.

C++

Every ten years or so another new method of programming comes along and we are told that this new wave will swamp the old ways and we had better get on the surf board or be left behind, soggy and unemployed. In the early sixties we were dragged kicking and screaming into Cobol and Fortran, all the time clutching desperately to that last beloved Autocoder listing. How we clung to those word marks, address registers, and index registers, believing that without them, programming would be nothing short of impossible. Just as our kicks atrophied and our screams faded and we succumbed, they wrested away our *goto*, invented the *while*, and shoved something called structured programming in our faces. Eventually, we came to love it — who wouldn't — but those changes were not easy to swallow because they required us to think of code structures in ways foreign to what we had come to practice as second nature.

Today we are told that the next tidal wave is "object-oriented programming," and I am trying to figure it out. The problem is whenever I read an explanation of what it means, the examples aren't real. Nowhere have I seen concrete examples of what objects are supposed to be. Good examples might be out there but I haven't found them yet. I can understand the given examples for what they are, but I can't yet transfer that understanding into what I design. Now, as I develop a new program, I ask myself which of the data aggregates should be objects and which should not and why. I ask but I do not answer.

We had better get this straight because something tells me that before long everyone else is going to be coding object-oriented languages and we are going to be left out. I was reluctant to admit this weakness in character and knowledge to a devoted readership until reading in Michael Swaine's column that the experts don't understand it either. Allow me to admit that my research has been restricted to magazine articles (going back several years to a *Byte* magazine feature issue on Smalltalk) and some product literature. If I work this enigma out, the C Column will try to explain to the rest of us what a few seem to already know. If this thing is truly the answer to a maiden's prayer, it sure could use a plain English explanation.

You might wonder why this column is the forum for such a wade in the dark through a murky new subject. I should be writing about things I already know about, right? Well, maybe a lot of you are facing the same confusion, and maybe a few of you are farther along. Perhaps the confused will benefit from this tentative entry into the unknown. Perhaps the aware will help.

Soon I will get a copy of Bjarne Stroustrup's book *The C++ Programming Language*, the definitive work on C++. C++ is a superset dialect of C that includes the constructs of object-oriented programming. You can define new classes of data in a manner similar to giving a *typedef* to a structure, but then you can bind functions to the new classes. Instances of a data class are called objects. New classes are derived from and inherit the attributes of old classes when you include the old class inside the new class. This feature is similar to the standard C ability to have one structure as a member of a higher structure.

The neat thing about it is that you can define operators that work with the objects. For example, if you need to add and subtract objects, you define that operator and provide the function that performs the operation. I'm not sure how much this feature is different from or better than the standard C facility for passing the addresses of two structures to a function. This faint explanation would be clearer if I could tell you what an obvious object was.

One feature of C++ is going to cause some trouble. C++ allows you to give the same name to several functions within a class. The compiler determines which one you want to call by matching the classes of the parameters in each call of the common name with the function prototypes. On the surface this seems to fly in the face of the strong typing claimed for C++, but then again I don't fully understand it yet.

Until recently C++ was implemented as a preprocessor that compiled into C language, which could then be compiled into executable code. This approach is how RATFOR was preprocessed into Fortran. Now a native C++ compiler for MS-DOS has been released. It is called the Zortech C++ Compiler and it is an offshoot of the recently disappearing Datalight Optimum C Compiler. The Datalight compiler was a respectable product that outperformed the competition with its speedy compile times until it was swept under by Turbo C and, later, QuickC.

Now, the talent that went into the Datalight development has been directed to the C++ world. I have a copy of the first release of the Zortech C++ Compiler, and I plan to use it to learn C++ and object-oriented programming if I can ever figure out what an object is. The Zortech manual has examples. Objects are toasters into which you can insert bread, set the darkness, and test for popped up toast. See what I mean? Sheesh! Another crotchet in the making. As I proceed with this quest for truth (computer science?) I will post my progress in the C Column. Write if you see an object that I might recognize.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

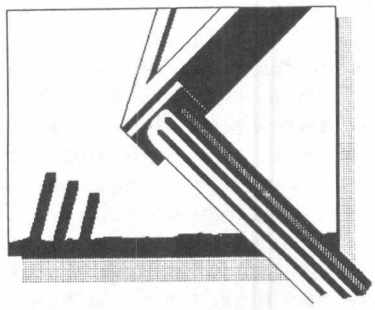
DDJ

(Listings begin on page 128.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

THE FORTH COLUMN

Wrap-up of Conferences



by Martin Tracy

Mark November 18 – 19 on your calendar and come to the Real-Time Programming Convention at the Grand Hotel in Anaheim, California (next to Disneyland). The invited guest speakers are Jef Raskin, head of the original Macintosh development team and inventor of the Canon Cat, and Ray Duncan, well-known author and expert on IBM PC operating systems. This convention is sponsored by the Forth Interest Group (FIG) but is not limited to Forth. If you are interested in software or hardware solutions for real-time problems, this show is for you. There will be talks and demonstrations on real-time operating systems, language-oriented Risc machines, parallel processing, and languages for data acquisition, analysis and real-time device control. Application areas will cover aerospace, medical, laboratory, machine-vision, digital signal processing, robotics, automation, intelligent instrumentation, adaptive devices, working neural nets, and software peripheral controllers.

Rochester Forth Conference

The day before the 1988 Rochester Forth Conference, Harris Semiconductor engineers, attired in railroad hats, held a one-day seminar in which they described the RTX 2000 Forth chip (see *DDJ* "Forth column" August 1988) and outlined their plans for its future. To everyone's surprise, at the conference itself, Phil Koopman (WISC Technologies) demonstrated a 32-bit hybrid of the RTX 2000 and WISC's own WCS technology. This chip is already in silicon! At a recent Silicon Valley FIG chapter, David Williams (Harris) announced that by this time next year, Harris expects to both double the speed of the RTX 2000 and come out with a 32-bit version.

Harris, makers of the RTX Forth Engine, announced that they have signed an agreement with Zoran, makers of Digital Signal Processing chips. In the agreement, Zoran will second-source the RTX chips and Harris will second-source Zoran's DSP devices. The Zoran chips will be able to be connected to the RTX ASIC bus for a very fast, high level language and digital signal processing system.

Also at the conference, Silicon Composers showed off their new SC/FOX RTX 2000 development system. The SC/FOX board slips into an IBM XT or AT (the hardware is smart) and opens up a dual-port 16K-RAM window anywhere in both the RTX 2000 and in the IBM address space. The board comes with 64K high-speed RAM, upgradeable to 1 Mbyte, and runs at 8 MHz, upgradeable to 10 MHz (approximately 15 Mips). It sells for $1,995. (Call 415-322-8763.) Both the Harris RTX 2000 development system and the SC/FOX use cross compilers, but neither — to my knowledge — has demonstrated its C compiler. The race to develop a stand-alone Forth for this chip is on.

The official theme of the Rochester Conference was Programming Environments, that took the form of idealized workstations and program libraries. Many Forths already support some kind of overlay mechanism. Jim Callahan (HS/FORTH), for example, talked about run-time dynamic linker to ASM and C libraries and Stephen Pelc described MicroProcessor Engineering's Modular Forth.

Guest speaker Dr. William Wickes spoke in detail on RPL (Reverse Polish Lisp) which he used to implement the symbolic math in HP's latest scientific pocket calculators. This fascinating hybrid looks more like Forth than Lisp and has the unusual property that definitions can contain embedded definitions. Dr. Norman Margolus, co-author of *Cellular Automata Machines* (reviewed in an earlier column), showed off the latest CAM board for high-speed modeling of physical fields. Mitch Bradley (Sun) talked about Forth's role in the Unix workstation environment — to bring Unix up in the first place and to provide for implementation-independent device drivers. There were almost 70 papers in all, and the proceedings should appear about the time you read this column.

ANS Forth Committee

The following letter was recently released by the ANS Forth committee:

"The Technical Committee working to develop an ANSI Standard for the Forth language is making substantial progress, according to Elizabeth D. Rather, Chair. The group, officially designated X3J14, has just concluded its fourth meeting in Rochester, NY (May 25 – 28).

"The most significant actions . . . have been passage of proposals to ensure the language won't be restricted to specific hardware architectures. . . . Recently passed pro-

posals facilitate transporting programs across CPUs of differing word lengths, and can even support one's complement architecture.

"Other major actions include adoption of a proposed extension defining an interface between Forth and standard operating systems such as MS-DOS, Unix, VMS, and OS/2. The group is also investigating the general areas of disk usage, control structures, and extensions such as floating-point arithmetic.

"The Technical Committee meets four times a year. Meetings remaining in 1988 are scheduled for October 25 – 29 at NASA's Goddard Space Flight Center, Greenbelt Maryland."

"To obtain a current copy of the working standard (BASIS), send $5 (payable to the Forth Vendors Group) to Martin Tracy, FORTH Inc., 111 N. Sepulveda Blvd. #300, Manhattan Beach, CA 90266."

New Products

Computer Continuum's LAB 40-Controller connects to the serial port of any personal computer and interfaces to their off-the-shelf modules for servo motor control, stepper motor drivers, 8- and 12-bit A/D, and relay control. The conversion of signals from RS-232 to their proprietary ribbon-cable bus is handled by a Motorola 68HC11 running a ROM'd Forth system. The controller itself includes a 64K memory map, 512 bytes of EPROM, eight channels of 8-bit 64 MHz A/D, three sockets for RAM or ROM, and RS-232, RS-422, or RS-485. Forth-based communication drivers are available for both the IBM and the Macintosh personal computers. The controller sells for $350 (415-755-1978).

The Dianax DSC-12 SBC is based on a 2-MHz R65F12 CPU. The board comes with D/A, A/D, signal conditioning and amplification, battery backup, and serial interface. The resident Forth nucleus has been extended with a full screen editor, assembler, and disassembler. Their introductory price of $499 includes an LCD display and 20-key keyboard. Call 514-878-2802 for more information.

FORTH Inc. (800-55-FORTH) announced the polyFORTH PC-based ACL development system. This 8086-based intelligent peripheral card provides eight RS-232 or RS-422 serial channels. Four boards can be multiplexed for a total of 32 channels of users or devices. The communication, monitoring, and control of each channel is offloaded to the ACL board, leaving the IBM PC polyFORTH system free for the application program.

More Math

In the last Forth column, there was a small bug in the floating-point support in the word ROUND. See the corrected definition in this month's Listing One, page 138.

Also, in a recent column I published Dr. Ting's recursive algorithm for line drawing. Wil Baden and Greg Bailey have both pointed out that real examples of useful recursive algorithms are hard to come by. Wil contributed the one in Listing Two, page 138.

If your Forth doesn't support RECURSE, try MYSELF. PARTITION finds the number of partitions of a number. *4 PARTITION* returns 5: 4 3+1 2+2 2+1+1 1+1+1+1.

Forth programmers often use the operators */ or */MOD to scale an integer number by a fractional amount. In other words: 1000 2 3 */ . prints 666, which is two thirds of 1000. The operator */ is preferred over the combination of * (multiplication) followed by / (division) since it supports a double-precision intermediate result.

For example, suppose you need to multiply a 16-bit integer or fixed-point fraction by pi (3.14159...). One approach is to multiply the number by 31416 and divide it by 10000, which is accurate to 4 digits. An alternate approach is to multiply by 355 and divide by 113, which is accurate to 7 digits. But how do you calculate the pair of numbers which is the best approximation to the desired constant?

Dr. Nathaniel Grossman's *Long Divisors and Short Fractions* (Forth Dimensions VI-3) discusses the mathematical procedure in readable detail. (I have shortened and rewritten his algorithm, which appears in Listing One.) His word CF (continued fraction) requires the double-number extensions UD* and UD/MOD which are in turn based on math primitives published in the previous column. The results of example 3.141592654 1000000000. CF is shown in Figure 1, this page.

355 113 is the largest 16-bit pair of integers available. Dr. Grossman points out, however, that the very next ratio is almost a 16-bit integer pair. Multiplication of unsigned 16-bit single numbers by pi can be approximated to good accuracy by

```
52174 33215 U*/ 2*
```

given the unsigned operator U*/ (or the polyFORTH operator */MOD).

PartQuot	Numerator	Denominator
3	3	1
7	22	7
15	333	106
1	355	113
293	104348	33215
11	1148183	365478
...	...	...
2	1570796327	500000000

Figure 1: *Sample continued fraction result*

Availability

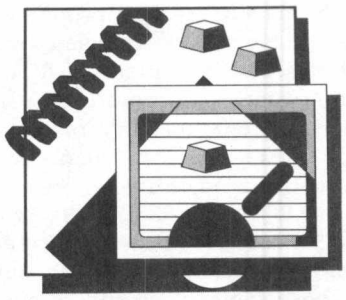
All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 138.)

Vote for you favorite feature/article.
Circle Reader Service **No. 8.**

On the Paradigm's Beat at Neural Nets



Ron, Kent, and I spent the end of July in San Diego at the 1988 IEEE International Conference on Neural Networks (IEEE ICNN-88). Many luminaries of neural-network research were there, including Bart Kosko (USC), Robert Hecht-Nielsen (HNC Inc. and UCSD), and Teuvo Kohonen (Helsinki University of Technology), all of whom were on the IEEE ICNN-88 conference committee. Marvin Minsky gave an opening talk in which he acknowledged what many neural-network people feel to be the case: that his critique of early work in the area derailed neural-net research for a decade. He hadn't intended his analysis to be prescriptive, he told the crowd. He was just trying to explain why people seemed to be leaving the field, not to encourage others to join them. His talk seemed well-received.

We attended the lectures (some of them), walked the exhibit floor, scoured the press room for press releases, bought the proceedings. We walked up and down the poster aisles, mystified. Anyway I was.

The poster sessions were a strange sight: rows of 4 × 8 boards set up at blackboard height, each covered with scribbled notes or typeset copy, or sequences of drawings, or photographs. In front of each board stood a speaker, the author of whatever is on the board, animatedly explaining the notes or copy or pictures to a small group of people, or waiting silently for an audience that may never form.

At the end of one poster aisle a priest stood in front of an empty board, talking in Italian to an audience of one. To say that it was incongruous would be to claim a better grasp of the congruity of the conference than I was ever able to secure, but it did seem odd.

by Michael Swaine

Too Narrow, Too Soon

The truth is, a lot of the poster sessions and other technical presentations went over my head. The conference had the feeling of an academic event. Some of the sessions could have been comprehensible and interesting only to a narrow subset of the already narrow set of attendees. Some presentations, though, took a broader perspective. There were dozens of presentations of algorithms for learning, an important issue in neural networks, and some of these offered some perspective. There was, for example, an overview of the back propagation learning technique by Paul Werbos, whose neuron-based back propagation approach was turned down as a thesis topic at Harvard in 1972. There were dozens of talks on network architectures.

There were also some applications talks. Kent kept asking the pragmatist's question: What can you do with this stuff? The answer that kept coming back was: Solve tomorrow's problems. Certainly a paradigm that requires a parallel architecture is not going to solve many of today's problems for the vast majority of computer users. Some of tomorrow's practical problems that people at the conference were using neural networks to solve were image processing, handwriting analysis, and speech recognition. The general level of all this work appears to be rudimentary.

There were also a dozen or so talks specifically classified as associative memory talks. Associative memory is a rich area of neural-network research.

And there were some presentations that made the attempt to map the frontier. Bart Kosko, in particular, presented a real map, a diagram showing the relationships among several of these memory models. It clears away a lot of underbrush to see that Hopfield Circuits and Brain-State-in-a-Box models are both additive, as opposed to multiplicative (a.k.a. shunting) models, and that these are all special cases of bidirectional associative memory (BAM) models, which are special cases of adaptive bidirectional associative memory (ABAM) models. As Ron pointed out to me, Kosko's style, in writing and speaking, is exceptionally concise and to the point.

Some of the presentations that were at the opposite end of the clarity scale from Kosko's may have suffered from the usual problem of focusing on results to the exclusion of a specification of assumptions. This is particularly a problem in a young paradigm whose assumptions and terminology may be rather slippery.

Of course, that's what the conference was for, to foster communication among neural-networks researchers and developers, to expand the base of common understanding that makes a paradigm a paradigm. On the basis of conversations overheard, a journalistic sense of how the attendees seemed to feel about the conference, and what I was able to get out of the conference myself, I'd say it was successful. Still, I had the sense that I was a graduate student again, puzzling over experimental designs and equations, and I wonder if that's not how a lot of the attendees felt (even if they didn't have to struggle as hard). And if so, I wonder if that's the right place for neural networks research to be. Is it getting too narrow too soon?

Semantic Vacuousness

The emphasis on technique also begs the question: What does it all mean? Neural networks is both an approach

to modeling the mind and a parallel-processing paradigm for programming. Psychological models need to refer to something psychological, but programming paradigms viewed just as programming paradigms don't have to mean anything. But I'm going to stick my neck out and say without any attempt to justify it that I think that the effectiveness of neural-network models, even viewed purely as a programming paradigm, will hinge in part on the semantics of the networks. What kind of semantics can we expect from neural networks?

Here's what Jerry Fodor, one of the leading thinkers on the philosophical foundations of cognitive research, has to say: "Rumor has it that, in semantics, AI is where the action is. But alas, soberly considered, computer models provide no semantic theory at all, if what you mean by a semantic theory is an account of the relation between language and the world."

A computer system—neural network, expert system, database, or whatever—that contains the element BOISE and the element CITY and the element IS-A may be able to construct something that we are pleased to call a representation of the fact that Boise is a city, but nothing internal to the system connects any part of it to the actual city Boise, or to the real-world fact. It looks entirely possible that we won't get a scientific semantics even in psychology; even less in programming. What we can get is not even an approximation to this; it's strictly symbols defined in terms of other symbols. What is that worth? How does it limit us? These seem to me to be hard questions that somebody ought to be asking.

A Semi-facetious Glossary

A conscientious student of neural networks with time to digest all the material from the conference would do well to present a glossary of neural-network terms. There are a lot of them, and the relationships among them are not always clear even when the terms have been defined. I am wrapping this up a couple of days after the conference, scrambling to meet a superstretched deadline, writ-

ing to fit. And I'm a dilettante, really. I can only, under the circumstances, give you a kind of cut-and-paste glossary, a list of some of the oft-heard terms, defined just about as the presenters defined them. What such a glossary shows has more to do with the state of communication in a developing discipline than with the useful clarification of fundamental terms.

The brief glossary that follows is not really intended to be useful. For that, it would have to be much longer, for one thing. And taking people's words out of context like this pretty much guarantees that the words will confuse rather that edify. My purpose in juxtaposing these definitions, with all the gaps in definition that they imply, is to demonstrate how people communicate in a developing paradigm. I'm acting here as a kind of anthropologist of programming.

Learning—Any change in any synapse. Some neural networks learn, some stabilize. Few do both.

Stability—The content addressable memory (CAM) property of neural networks, the input ball rolling down the nearest attractor basin, dissipating energy as it rolls.

Backpropagation—A supervised learning algorithm, complete with particular choice of unit transfer functions (e.g., semi linear with logistic squashing), error function (e.g., mean-square error), and weight update rule (e.g., using momentum).

Backpropagation—A learning algorithm for multi-layer feedforward networks.

Boltzmann machine (BM)—A parallel computing network consisting of simple processing units connected by bidirectional links.

Associative memory (AM)—A process dual to self-organized pattern formation in nonlinear dynamical systems.

Bidirectional associative memory (BAM)—An associative memory that uses forward and backward bidirectional search to recall an associated bipolar vector pair from an input pair.

Bidirectional associative memory (BAM)—A minimal two-layer nonlinear feedback network.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Listing One *(Text begins on page 108.)*

```
/* ---------- menu.h ---------- */

typedef struct w_menu {
            char *mname;                                /* menu bar selection names                                  */
            char *mhlpmsg;                              /* menu bar prompting messages                               */
            char **mselcs;                              /* the pop-down menu selections                              */
            char **mshelp;                              /* help mnemonics for the pop-down selections   */
            char *mskeys;                               /* key strokes that accompany the selections    */
            int (**func)(int,int);          /* the functions to execute for the selections  */
            int lastvsel;                               /* most recent vertical selection                            */
} MENU;

void menu_select(MENU *, int);
char *display_menubar(MENU *);
void restore_menubar(char *);
```

End Listing One

Listing Two

```
/* ------------ menu.c ------------ */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "window.h"
#include "menu.h"

#define ON 1
#define OFF 0

static int getvmn(void);
static void haccent(int);
static void dimension(char **,int *,int *);
static void light(int);
static int vlook(int,int);

int hsel = 1;           /* horizontal selection */
MENU *mn;               /* active menu          */

/* ------------- display & process a menu ----------- */
void menu_select(MENU *mnn, int sel)
{
    int hs, sx, sy, vsel, holdhsel, frtn = FALSE;
    char *mb;

    if (sel)
        hsel = sel;
    mn = mnn;
    sx = wherex();
    sy = wherey();
    mb = display_menubar(mn);
    light(ON);
    while (!frtn && ((vsel = getvmn()) != 0))    {
        light(OFF);
        holdhsel = hsel;
        hs = hsel;
        hsel = 1;
        frtn = (*(mn+hs-1)->func [vsel-1]) ?
               (*(mn+hs-1)->func [vsel-1])(hs,vsel) : FALSE;
        hsel = holdhsel;
        mn = mnn;
        light(ON);
    }
    light(OFF);
    gotoxy(sx, sy);
    restore_menubar(mb);
}

/* --- display the menu bar with no selections chosen ---- */
char *display_menubar(MENU *mn)
{
    int i = 0;
    char *mb;

    if ((mb = malloc(160)) != NULL)
        gettext(1,1,80,1,mb);
    window(1,1,80,25);
    gotoxy(1,1);
    textcolor(MENUFG);
    textbackground(MENUBG);
    cprintf("    ");
    while ((mn+i)->mname)
        cprintf(" %-10.10s ", (mn+i++)->mname);
    while (i++ < 6)
        cprintf("            ");
    cprintf("    ");
    hidecursor();
    return mb;
}

/* ------------ restore the menu bar line --------------- */
void restore_menubar(char *mb)
{
    if (mb)    {
        puttext(1,1,80,1,mb);
        free(mb);
    }
}

/* ---------pop down a vertical menu --------- */
static int getvmn()
{
    int ht, wd, vx, sel;

    while (TRUE)    {
        dimension((mn+hsel-1)->mselcs, &ht, &wd);
        if (!(mn+hsel-1)->lastvsel)
            (mn+hsel-1)->lastvsel = 1;
        if (ht > 0)    {
            vx = 5+(hsel-1)*12;
            establish_window(vx, 2, vx+wd+1, ht+3,
                MENUFG, MENUBG,TRUE);
            text_window((mn+hsel-1)->mselcs, 1);
            sel = select_window((mn+hsel-1)->lastvsel,
                SELECTFG, SELECTBG, vlook);
            delete_window();
            if (sel == FWD || sel == BS)
                haccent(sel);
            else
                return ((mn+hsel-1)->lastvsel = sel);
        }
        else    {
            if ((sel = getkey()) == *(mn+hsel-1)->mskeys)
                return 1;
            switch (sel)    {
                case FWD:
                case BS:    haccent(sel);
                            break;
                case '\r':    return 1;
                case ESC:    return 0;
                default:    putch(BELL);
                            break;
            }
        }
    }
}

/* ---- if vertical menu user types FWD, BS,
                or a menu key, return it ---- */
static int vlook(ch, sel)
{
    char *cs, *ks;

    if (ch == FWD || ch == BS)    {
        (mn+hsel-1)->lastvsel = sel;
        return ch;
    }
    ks = (mn+hsel-1)->mskeys;
    if ((cs = memchr(ks, tolower(ch), strlen(ks))) == NULL)
        return ERROR;
    return ((mn+hsel-1)->lastvsel = cs-ks+1);
}

/* ----- manage the horizontal menu selection accent ----- */
static void haccent(int sel)
{
    switch (sel)    {
        case FWD:
            light(OFF);
            if ((mn+hsel)->mname)
                hsel++;
            else
                hsel = 1;
            light(ON);
            break;
        case BS:
            light(OFF);
```

CIRCLE NO. 106 ON READER SERVICE CARD

```
            if (hsel == 1)
                while ((mn+hsel)->mname)
                    hsel++;
            else
                --hsel;
            light(ON);
            break;
        default:
            break;
    }
}

/* ---------- compute a menu's height & width --------- */
static void dimension(char *sl[], int *ht, int *wd)
{
    *ht = *wd = 0;

    while (sl && sl [*ht])    {
        *wd = max(*wd, (unsigned) strlen(sl [*ht]));
        (*ht)++;
    }
}

/* --------- accent a horizontal menu selection ---------- */
static void light(int onoff)
{
    extern struct wn wkw;
    extern char spaces[];
    int ln;

    window(1,1,80,25);
     textcolor(onoff ? SELECTFG : MENUFG);
     textbackground(onoff ? SELECTBG : MENUBG);
     gotoxy((hsel-1)*12+6, 1);
    cprintf((mn+hsel-1)->mname);
    textcolor(TEXTFG);
    textbackground(TEXTBG);
    if ((mn+hsel-1)->mhlpmsg)    {
        if (onoff)    {
            ln = strlen((mn+hsel-1)->mhlpmsg);
            gotoxy((80-ln)/2,25);
            cprintf((mn+hsel-1)->mhlpmsg);
        }
        else    {
            gotoxy(1,25);
            cprintf(spaces);
        }
    }
    current_window();
    hidecursor();
}
```

End Listing Two

Listing Three

```
/* --------- testmenu.c ------------ */
#include <stdio.h>
#include <string.h>
#include "window.h"
#include "menu.h"

/* ---------- menu tables --------- */
static char *fselcs[] = {
            "Load",
            "Save",
            "New",
            "Quit    [Esc]",
            NULL
};

static char *eselcs[] = {
            "Move        [F3]",
            "Copy        [F4]",
            "Delete      [F8]",
            "Find        [F7]",
            NULL
};

static char *oselcs[] = {
            "Auto Paragraph Reformat        ",
            "Insert  [Ins]                  ",
            NULL
};

static int quit(int,int);
static int reform(int, int);
static int insert(int, int);
static void set_toggles(void);

static char forced[] = {F3,F4,F8,F7};

static char options[] = {'a', INS};

static int (*ffuncs[])()={NULL,NULL,NULL,quit};
static int (*efuncs[])()={NULL,NULL,NULL,NULL};
static int (*ofuncs[])()={reform,insert};

static char filehelp[] =
   "Load a file, Save current buffer, Type a new file, Quit";
static char edithelp[] =
   "Move, Copy, Delete blocks, Find a string";
static char optnhelp[] =
   "Toggle editor options";

MENU emn [] = {
   {"File",    filehelp, fselcs, NULL,   "",      ffuncs, 0},
   {"Edit",    edithelp, eselcs, NULL,   forced,  efuncs, 0},
   {"Options", optnhelp, oselcs, NULL,   options, ofuncs, 0},
   {NULL}
};

void main()
{
            set_toggles();
            clear_screen();
            menu_select(emn, 1);
            clear_screen();
}

/* ---- illustrates toggled menu mode selectors -------- */
int reforming, inserting;  /* these are mode variables */

static int reform(hs,vs)
{
            reforming ^= TRUE;
            set_toggles();
            return FALSE;
}

static int insert(hs,vs)
{
            inserting ^= TRUE;
            set_toggles();
            return FALSE;
}

static void set_toggles()
{
            strcpy(&oselcs[0][24], reforming ? "(on) " : "(off)");
            strcpy(&oselcs[1][24], inserting ? "(on) " : "(off)");
}

/* ---- when TRUE is returned, the menu system exits ----- */
static int quit(hs,vs)
{
            return TRUE;
}
```

End Listing Three

Listing Four

```
/* --------- entry.h ---------- */

typedef struct field {                    /* data entry field description*/
            int frow;                                 /* field row                              */
            int fcol;                                 /* field column position          */
            int             fx;                                                    /* running column       */
            char *fbuff;                              /* field buffer                           */
            char *fmask;                              /* field data entry mask          */
            char *fhelp;                              /* field help window mnemonic  */
} FIELD;

void field_tally(void);
int data_entry(FIELD *, int, int);
void clear_template(void);
void insert_line(void);

#define INSERTING TRUE                    /* initial Insert mode */
```

End Listing Four

Listing Five

```
/* --------- entry.c ---------- */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#include "window.h"
#include "entry.h"

#define FIELDCHAR '_'
int inserting = INSERTING;      /* insert mode, TRUE/FALSE */

extern struct wn wkw;

/* -------- local prototypes -------- */
static void addfield(FIELD *);
static void disp_field(FIELD *, char *, char *);
static int read_field(int);
static void data_value(FIELD *);
static int endstroke(int);
static void home_cursor(void);
static int backspace(void);
static void end_cursor(void);
static void forward(void);
static int fore_word(void);
static int back_word(void);
static void delete_char(void);
static void delete_word(void);

static FIELD *fhead;
static FIELD *ftail;
FIELD *fld;
/* -------- display a data field ------ */
static void disp_field(FIELD *fldv, char *bf, char *msk)
{
    char cl[80], *cp = cl;

    while (*msk)    {
        *cp++ = (*msk != FIELDCHAR ? *msk : *bf++);
        msk++;
    }
    *cp = '\0';
    writeline(fldv->fx + fldv->fcol - 1,
              fldv->frow, cl);
}

/* ------- display the data value in a field ------ */
static void data_value(FIELD *fldv)
{
    gotoxy(fldv->fcol, fldv->frow);
    fldv->fx = 1;
    disp_field(fldv, fldv->fbuff, fldv->fmask);
}

/* ------ display all the fields in a window ------- */
void field_tally()
{
    FIELD *fldt;

    fldt = fhead;
    while (fldt != ftail)    {
        data_value(fldt);
        fldt++;
    }
}
/* ------- clear a template to all blanks ------ */
void clear_template()
{
    FIELD *fldc;
    char *bf, *msk;

    fldc = fhead;
    while (fldc != ftail)    {
        bf = fldc->fbuff;
        msk = fldc->fmask;
        while (*msk)    {
            if (*msk == FIELDCHAR)
                *bf++ = ' ';
            msk++;
        }
        fldc++;
    }
    *bf = '\0';
    field_tally();
}

char *mask, *buff;

/* ------- read a field from the keyboard ------------- */
static int read_field(c)
{
    int done = FALSE, first = TRUE;

    home_cursor();
    while (TRUE)    {
        gotoxy(fld->fx+fld->fcol-1, fld->frow);
        if (!c || !first)
            c = getkey();
        first = FALSE;
        switch (c)    {
            case CTRL_D:
                delete_word();
                break;
            case CTRL_FWD:
                done = !fore_word();
                break;
            case CTRL_BS:
                done = !back_word();
                break;
            case HOME:
                fld->fx = 1;
                home_cursor();
                break;
            case END:
                end_cursor();
                break;
            case FWD:
                forward();
                break;
            case BS:
                backspace();
                break;
            case '\b':
                if (!backspace())
                    break;
                gotoxy(fld->fx+fld->fcol-1, fld->frow);
            case DEL:
                delete_char();
                disp_field(fld, buff, mask);
                break;
            case INS:
                inserting ^= TRUE;
                insert_line();
                break;
            default:
                if (endstroke(c))    {
                    done = TRUE;
                    break;
                }
                if (inserting)    {
                    memmove(buff+1, buff, strlen(buff)-1);
                    disp_field(fld, buff, mask);
                    gotoxy(fld->fcol+fld->fx-1, fld->frow);
                }
                *buff = (char) c;
                putch(c);
                forward();
                if (!*mask)
                    c = FWD;
                break;
        }
        if (done || !*mask)
            break;
    }
    wkw.wx = fld->fx+1;
    return c;
}
/* -------- home the cursor in the field -------- */
static void home_cursor()
{
    buff = fld->fbuff+fld->fx-1;
    mask = fld->fmask+fld->fx-1;
    while (*mask != FIELDCHAR)    {
        fld->fx++;
        mask++;
    }
}

/* ------- move the cursor to the end of the field ------ */
static void end_cursor()
{
    fld->fx += strlen(mask)-1;
    buff += strlen(buff)-1;
    mask += strlen(mask)-1;
    while (*buff == ' ')
        if (!backspace())
            break;
    forward();
}

/* ------ move the cursor forward one character -------- */
static void forward()
{
    do    {
        fld->fx++;
        mask++;
    } while (*mask && *mask != FIELDCHAR);
    buff++;
}

/* ------- move back one character ------------ */
static int backspace()
{
    if (buff != fld->fbuff)    {
        --buff;
        do    {
            --mask;
            --(fld->fx);
```

```
        } while (*mask != FIELDCHAR);
        return TRUE;
    }
    return FALSE;
}

/* ------- move forward one word ------- */
static int fore_word()
{
    int ct = 2, test = *buff == ' ';

    while (ct--)    {
        while ((*buff == ' ') == test && *mask)
            forward();
        if (!*mask)
            return FALSE;
        if (test)
            break;
        test = !test;
    }
    return TRUE;
}

/* ------- move backward one word ------- */
static int back_word()
{
    int test;

    if (buff == fld->fbuff)
        return FALSE;
    if (*(buff-1) == ' ')
        backspace();
    test = *buff == ' ';
    while ((*buff == ' ') == test && buff != fld->fbuff)
        backspace();
    if (test)
        while (*buff != ' ' && buff != fld->fbuff)
            backspace();
    if (*buff == ' ')
        forward();
    return TRUE;
}

/* --------- delete a character ----------- */
static void delete_char()
{
    memmove(buff, buff+1, strlen(buff));
    *(buff+strlen(buff)) = ' ';
}

/* ----------- delete a word ---------- */
static void delete_word()
{
    int test = *buff == ' ';
    int ln = strlen(buff);

    while ((*buff == ' ') == test && ln--)
        delete_char();
    if (!test)
        delete_char();
    disp_field(fld, buff, mask);
}

/* ---------- test c for an ending keystroke ----------- */
static int endstroke(int c)
{
    switch (c)    {
        case '\r':
        case '\n':
        case '\t':
        case ESC:
        case F1:
        case F2:
        case F3:
        case F4:
        case F5:
        case F6:
        case F7:
        case F8:
        case F9:
        case F10:
        case PGUP:
        case PGDN:
        case HOME:
        case END:
        case CTRL_FWD:
        case CTRL_BS:
        case CTRL_HOME:
        case CTRL_END:
        case UP:
        case DN:
            return TRUE;
        default:
            return FALSE;
    }
}

/* ----- Process data entry for a screen template. ---- */
int data_entry(FIELD *fldin, int init, int firstfld)
{
    int exitcode = 0, done = FALSE;
    insert_line();
    fhead = ftail = fld = fldin;
    while (ftail->frow)
        ftail++;
    while (firstfld-- && fld != ftail)
        fld++;
    --fld;
    if (init)
        clear_template();
    field_tally();
    /* ---- collect data from keyboard into screen ---- */
    while (done == FALSE)    {
        gotoxy(fld->fcol, fld->frow);
        textcolor(FIELDFG);
        textbackground(FIELDBG);
        data_value(fld);
        gotoxy(fld->fcol, fld->frow);
        fld->fx = 1;
        exitcode = read_field(0);
        textcolor(ENTRYFG);
        textbackground(ENTRYBG);
        data_value(fld);
        switch (exitcode)    {
            case DN:
            case '\r':
            case '\t':
            case CTRL_FWD:
            case FWD:    done = (ftail == fhead+1);
                         fld++;
                         if (fld == ftail)
                             fld = fhead;
                         break;
            case CTRL_BS:
            case UP:     if (fld == fhead)
                             fld = ftail;
                         --fld;
                         break;
            case CTRL_HOME:
                         fld = fhead;
                         break;
            case CTRL_END:
                         fld = ftail-1;
                         break;
            default:     done = endstroke(exitcode);
                         break;
        }
    }
    fld = NULL;
    return (exitcode);
}

/* ---------- set insert/exchange cursor shape ---------- */
void insert_line()
{
    set_cursor_type(inserting ? 0x0106 : 0x0607);
}
```

End Listing Five

Listing Six

```
/* ---------- testentr.c --------- */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "window.h"
#include "entry.h"

struct config {
    char name[36];
    char acctno[21];
    char password[11];
    char phone[15];
} cfg;

static char mask[] = "____________________________________";
static char phmask[] = "(___) ___-____ ext.____";

FIELD pers_template[] = {
    {3, 14, 1, cfg.name,     mask,     NULL},
    {4, 14, 1, cfg.acctno,   mask+15,  NULL},
    {5, 14, 1, cfg.password, mask+25,  NULL},
    {6, 14, 1, cfg.phone,    phmask,   NULL},
    {0}
};

void main()
{
    establish_window(15,5,65,12,ENTRYFG,ENTRYBG,TRUE);
    window_title(" Personal Data ");
    gotoxy(3,3);
    cputs("Name:");
    gotoxy(3,4);
    cputs("Account #:");
    gotoxy(3,5);
    cputs("Password:");
    gotoxy(3,6);
    cputs("Phone:");
    data_entry(pers_template, TRUE, 1);
    delete_window();
    return FALSE;
}
```

End Listings

Listing One *(Text begins on page 100.)*

```
' Program HUGEMATS.BAS
' Demo program to add two huge matrices > 64K, giving a third
' Written using Microsoft QuickBasic 4.00B
' Kent Porter, DDJ, October 1988

DEFINT A-Z                      ' All variables are integers
DECLARE SUB acquire (D())       ' Subroutine prototype
REM $DYNAMIC                    ' Use heap for arrays

' Constants
CONST maxRows = 250             ' Rows in matrices
CONST maxCols = 300             '  and columns

' Define arrays
OPTION BASE 1                   ' 1 is lowest subscript
DIM A(maxRows, maxCols)
DIM B(maxRows, maxCols)
DIM C(maxRows, maxCols)

' ------------------------------------------------------------
' Main program follows
  CLS                           ' Clear screen
  size& = maxRows * 2
  size& = size& * maxCols       ' Array size as long int
  PRINT "Size of each array is"; size&; "bytes"

  PRINT "Setting up Array A"
  Acquire A()

  PRINT "Setting up Array B"
  Acquire B()

  PRINT "Adding arrays"
  FOR col = 1 TO maxCols
    FOR row = 1 TO maxRows
      C(row, col) = A(row, col) + B(row, col)
    NEXT row
  NEXT col

  PRINT "Proof:"
  PRINT "A(1, 1) + B(1,1) = C(1, 1) = ";
  PRINT A(1, 1); " + "; B(1, 1); " = "; C(1, 1)
  C = maxCols
  r = maxRows
  PRINT "A(max, max) + B(max, max) = C(max, max) = ";
  PRINT A(r, C); " + "; B(r, C); " = "; C(r, C)
  ------------------------------------------------------------

SUB Acquire (D())
  ' Load data into array 'D'

  FOR row = 1 TO maxRows
    FOR col = 1 TO maxCols
      D(row, col) = (row * 10) + col     ' Generate test data
    NEXT col
  NEXT row
END SUB
```

End Listing One

Listing Two

```
PROGRAM DiskArr;
(* Illustrates disk-based arrays, adding two 500 x 500 arrays *)
(*   of REAL to yield a third.                                *)
(* Requires 4.5MB of disk space                               *)
(* Turbo Pascal 4.0                                           *)
(* Kent Porter, DDJ, October 1988                             *)

USES CRT, DOS;

CONST  maxRow = 499;
       maxCol = 499;
       Yes    = TRUE;
       No     = FALSE;

TYPE   ArrayRow = ARRAY [0..MaxCol] OF REAL;     (* Row buffer *)
       RowFile  = FILE OF ArrayRow;               (* File type *)
       BuffCtlBlock = RECORD       (* Row buffer control block *)
         CurrentRow : WORD;
         IsModified : BOOLEAN;
       END;

VAR    ArrA, ArrB, ArrC  : RowFile;
       RowA, RowB, RowC  : ArrayRow;
       BufA, BufB, BufC  : BuffCtlBlock;
       BufSize           : WORD;
       row, col, nCols   : WORD;

(* ------------------------------------------------------------ *)

PROCEDURE Acquire (VAR arr  : RowFile;
                   VAR cb   : BuffCtlBlock;
                   VAR buf  : ArrayRow;
                       name : String);

  (* Load data into disk array 'arr'                             *)
  (* If the file already exists, simply open it                  *)
  (* Upon return, row 0 is loaded into the buffer                *)

VAR   r, c, nread : WORD;
      newfile     : BOOLEAN;

BEGIN
  cb.CurrentRow := 0;       (* Initialize buffer control block *)
  cb.IsModified := No;
  NewFile       := Yes;     (* Assume we have to make new file *)

  Assign (arr, name);
  {$I-}
  Reset (arr);                           (* Does the file exist? *)
  {$I+}
  IF IOResult = 0 THEN                    (* File already exists *)
    IF FileSize (arr) = maxRow+1 THEN          (* If right size *)
      NewFile := No;                  (* then use existing file *)

  (* If we have to create a new file *)
  IF NewFile THEN BEGIN
    Rewrite (arr);                              (* Create the file *)
    FOR r := 0 TO maxRow DO BEGIN
      Gotoxy (1, WhereY-1); Writeln ('Row ',r:3); (* Show row *)
      FOR c := 0 TO maxCol DO
        Buf [c] := ((row * nCols) + c) * 1.0;     (* Test data *)
      Write (arr, buf);                  (* Write out full row *)
    END;
    Writeln;
  END;

  Seek (arr, 0);                             (* Go to top of file *)
  Read (arr, buf);                             (* Get first block *)
END;
(* -------------------------- *)

FUNCTION A (row, col : WORD) : REAL;

  (* Return indicated element from Array A *)

BEGIN
  IF row <> BufA.CurrentRow THEN BEGIN      (* Reading new row *)
    IF BufA.IsModified THEN BEGIN      (* Save row if modified *)
      Seek (ArrA, LONGINT (BufA.CurrentRow));
      Write (ArrA, RowA);
    END;
    Seek (ArrA, LONGINT (row));                    (* Get new row *)
    Read (ArrA, RowA);
    BufA.IsModified := No; BufA.CurrentRow := row;
  END;
  A := RowA [col];                          (* Return the element *)
END;
(* -------------------------- *)

FUNCTION B (row, col : WORD) : REAL;

  (* Same as A, but from ArrB *)

BEGIN
  IF row <> BufB.CurrentRow THEN BEGIN
    IF BufB.IsModified THEN BEGIN
      Seek (ArrB, LONGINT (BufB.CurrentRow));
      Write (ArrB, RowB);
    END;
    Seek (ArrB, LONGINT (row));
    Read (ArrB, RowB);
    BufB.IsModified := No; BufB.CurrentRow := row;
  END;
  B := RowB [col];
END;
(* -------------------------- *)

FUNCTION C (row, col : WORD) : REAL;

  (* Same as A, but from ArrC *)

BEGIN
  IF row <> BufC.CurrentRow THEN BEGIN
    IF BufC.IsModified THEN BEGIN
      Seek (ArrC, LONGINT (BufC.CurrentRow));
      Write (ArrC, RowC);
    END;
    Seek (ArrC, LONGINT (row));
    Read (ArrC, RowC);
    BufC.IsModified := No; BufC.CurrentRow := row;
```

```
  END;
  C := RowC [col];
END;
(* -------------------------- *)

PROCEDURE WriteToC (row, col : WORD; val : REAL);

  (* Write val to C [row, col] *)

BEGIN
  IF row <> BufC.CurrentRow THEN BEGIN          (* If a new row *)
    IF BufC.IsModified THEN BEGIN             (* and old changed *)
      Seek (ArrC, LONGINT (BufC.CurrentRow));      (* save old *)
      Write (ArrC, RowC);
    END;
    Seek (ArrC, LONGINT (row));               (* then get new row *)
    Read (ArrC, RowC);
    BufC.CurrentRow := row;
  END;
  RowC [col] := val;                           (* and write to it *)
  BufC.IsModified := Yes;
END;
(* -------------------------- *)

BEGIN   (* Body of main program *)
  ClrScr;
  Writeln ('*** Disk Array Processor ***');
  nCols := MaxCol + 1;
  BufSize := SizeOf (ArrayRow);

  (* Create output array file and fill with zeros *)
  Assign (ArrC, 'ARRAY.C');
  Rewrite (ArrC);
  Writeln ('Initializing output array'); Writeln;
  FOR col := 0 TO maxCol DO
    RowC [col] := 0.0;
  FOR row := 0 TO maxRow DO BEGIN
    Gotoxy (1, WhereY-1); Writeln ('Row ', row:3);
    Write (ArrC, RowC);
  END;
  Seek (ArrC, 0); Read (ArrC, RowC);
  BufC.CurrentRow := 0; BufC.IsModified := No;

  (* Get the test data into A and B *)
  Gotoxy (1, WhereY-1); Writeln ('Setting up Array A');
  Acquire (ArrA, BufA, RowA, 'ARRAY.A');
  Gotoxy (1, WhereY-1); Writeln ('Setting up Array B');
  Acquire (ArrB, BufB, RowB, 'ARRAY.B');
  (* Add A and B, giving C *)
  Gotoxy (1, WhereY-1); ClrEol; Writeln ('Adding arrays');
  FOR row := 0 TO maxRow DO BEGIN
    Gotoxy (1, WhereY);
    Write ('Row ', row:3);
    FOR col := 0 TO maxCol DO
      WriteToC (row, col, (A (row, col) + B (row, col)));
  END;

  (* Display proof that it worked *)
  Gotoxy (1, WhereY); Writeln ('Addition completed');
  Writeln ('Proof:');
  Write   ('A (1, 1) + B (1, 1) = C (1, 1) = ');
  Writeln (A (1, 1):6:0, ' + ',
           B (1, 1):6:0, ' = ',
           C (1, 1):6:0);

  Write   ('A (maxRow, maxCol) + B (maxRow, maxCol) = ');
  Writeln ('C (maxRow, maxCol) = ');
  Writeln (A (maxRow, maxCol):6:0, ' + ',
           B (maxRow, maxCol):6:0, ' = ',
           C (maxRow, maxCol):6:0);
  Close (ArrC);
END.
```

End Listings

THE FORTH COLUMN

Listing One *(Text begins on page 116.)*

```
\ Correction for August 88 Forth column:
: ROUND ( t - ud exp )   \ assumes hi bit is zero.
   MIN# 0 0 T+  ROT DROP  DUP 0< DUP IF  DROP  DU2/ 1  THEN ;

( Recursive procedure to find the number of partitions of)
( a number.  E.g.  4 has 5 partitions: 4 3+1 2+2 2+1+1 1+1+1+1)
: SUBPARTITION
( m n -- p: # of partitions of m of size at most n)
   OVER MIN   DUP 1 =
   IF   ( m 1)
        2DROP 1   ( 1)
   ELSE ( m n)
        2DUP =
   IF   ( m m)
        1-   RECURSE ( p)  1+
   ELSE ( m n)
        2DUP 1- ( m n m n-1)  RECURSE ( m n p)
        ROT ROT ( p m n) DUP >R  - R> ( p m-n n)
        RECURSE ( p p')  +
   THEN THEN ;

: PARTITION ( n - p: Number of partitions of n.)
   DUP 0> NOT ABORT" Argument must be positive."
   DUP SUBPARTITION ;

\ Double-precision multiplication.
: UD* ( ud ud2 - ud3)
   ROT 2>R  2DUP UM*  ROT R> UM* DROP +  ROT R> UM* DROP + ;

\ Double-precision division.
: UD/MOD ( ud ud2 - udr udq)   ?DUP   \ double unsigned /MOD
\ Adapted from Dr. Grossman, FD VI-3.  Correction by Bob Royce.
   IF  DUP 1+ ( not SuperBASE-1?)
     IF  DUP  0 1 ( SuperBASE) ROT 1+ UM/MOD  >R DROP
       2DUP R@ UT* DROP ( scale ud2)
       2ROT DUP 0 2ROT           DUP 2>R  UT*  R> UT/
       2OVER 2SWAP D-  2R> >R UT*
     ELSE  2DUP 2ROT DUP 0 2ROT  DUP 2>R  UT*  R> UT/
       2OVER 2SWAP D-  0
     THEN  R> UT/  >R DROP  2SWAP  R@ UT* DROP  D-  R> 0
   ELSE  >R  2DUP 0 R@ UT/
       2SWAP 2OVER  R> UT* DROP  D- 2SWAP
   THEN ;

\ Decomposition by continued fractions -- long version.
2VARIABLE B_BIN  2VARIABLE Q_BIN  2VARIABLE R_BIN

: CF ( d d2)   \ Adapted from Dr. Grossman, FD VI-3.
\ prints continued fraction decomposition of d/d2.
   CR  ." Partial_Quot    Numerator Denominator"
   1 0  0 0  2ROT 2>R 2ROT 2R>  0 0  1 0  2ROT 2>R 2ROT 2R>
   BEGIN  2SWAP 2OVER UD/MOD
     CR  2DUP 12 D.R  B_BIN 2!  R_BIN 2!  Q_BIN 2!
     2ROT 2>R 2ROT 2R>  2>R 2>R 2OVER 2OVER 2R> 2R>
     2ROT 2>R 2ROT 2R>       B_BIN 2@  UD*  2ROT D+
     2DUP 13 D.R  2ROT 2ROT B_BIN 2@  UD*       D+
     2DUP 12 D.R  2SWAP Q_BIN 2@ R_BIN 2@  2DUP D0=
   UNTIL  2DROP 2DROP 2DROP  2DROP 2DROP 2DROP  ;
```

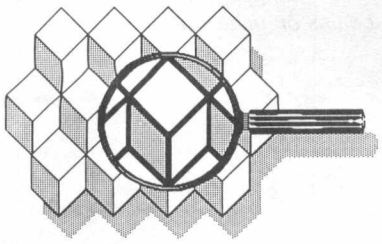
End Listing One

Listing Two

```
: SUBPARTITION ( m n - p)
   OVER MIN  DUP 1 =
   IF  2DROP 1
   ELSE  2DUP =
     IF  1-  RECURSE  1+
     ELSE  2DUP 1-  RECURSE
       ROT ROT  DUP >R  - R>
       RECURSE  +
   THEN THEN ;

: PARTITION ( n - p)
   DUP 0> NOT
   ABORT" Must be positive"
   DUP  SUBPARTITION ;
```

End Listings

OF INTEREST



Programming Tools

Dis*Doc, a new automated disassembler and patcher for the IBM PC, XT, AT, and compatibles, has been released by **RJ Swantek & Associates**. Dis*Doc uses four algorithms to separate code from data at a rate of 10,000 lines per minute on a hard disk.

It also processes the 80386/80387 instruction set; disassembles anything in files or RAM memory; patches any file using the same addresses given in the disassembled listing; creates a MASM ready output, including program declarations; automatically generates five different labels from Jump, Branch, Call, Data, Stack, or allows programmers to create their own; automatically generates comments for MS-DOS and BIOS services as well as I/O commands; keeps patches in a separate file for documentation purposes; saves the results of its first program pass to expedite the creation of future listings; and has error messages that tell the programmer how to fix the problem.

Selling for $99.95 plus $4 shipping and handling, Dis*Doc requires two floppy drives, or a hard disk and floppy drive, and at least 256K bytes of memory for MS-DOS or PC-DOS 2.0 or higher. Reader Service No. 23.
RJ Swantek
P.O. Box 1032
Hartford, CT 06111
203-560-0236

SLR Systems has announced OPTASM, an optimizing 80x86 assembler. It is fully (except for 386) source compatible with MASM 5.0 and includes full support for Microsoft's CodeView debugger. Version 1.5 also features comprehensive on-line help. It interfaces with most high-level languages including Borland's Turbo Pascal, Turbo C, and Microsoft's C 5.1.

This product requires DOS 2.0 or greater and 256K of available memory; a hard disk is recommended. OPTASM, Version 1.5 sells for $125; upgrades are available to currently registered users of version 1.0 for $29.95. Reader Service No. 38.
SLR Systems
1622 N Main St.
Butler, PA 16001
412-282-0864

Ready Systems has introduced VRTX32, a real-time operating system for the Intel 80386 microprocessor. VRTX32 includes a real-time multitasking kernel. Ready Systems also announced RTscope, a real-time debugger which includes a VRTX32 system monitor. VRTX32/386 is a real-time executive designed for use as a standard software component in security applications. It provides comprehensive support for the 386 architecture, specifically, full 32-bit addressing.

RTscope monitors VRTX32's system objects (such as queues and tasks) while operating concurrently with the multitasking system. It also performs traditional debugging functions such as displaying and setting 386 register and memory functions. RTscope operates in both tasking and command mode, and it displays the contents of local and global descriptor tables (LDTs and GDTs), and can set and remove 386 hardware breakpoints.

The R&D package sells for $6,775 for VRTX32/386 and $3,000 for RTscope 386. Reader Service No. 27.
Ready Systems
449 Sherman Ave.
Palo Alto, CA 94306
415-326-2950

Programmer's Library, a new CD-ROM product by **Microsoft**, gives programmers on-line access to a comprehensive collection of books, technical manuals, and sample programs.

The 48 books and technical manuals in the library provide information on Microsoft's operating systems and languages, ranging from quick reference help to detailed discussions. The disk also includes several references for hardware devices, including the Microsoft Mouse, CD-ROM drives, and video cards, as well as 7 Mbytes of sample code. All material can be accessed from inside the user's text editor or word processor and copied into programs or documents without rekeying.

Suggested price is $395 which includes one free update to be available in early 1989. Reader Service No. 28.
Microsoft
16011 NE 36th Way
Redmond, WA 98073-9717
206-882-8080

The Software Bottling Company has released SoftCode, a new type of program generator that allows programmers to generate any type of code in any programming language. SoftCode comes complete with editor and program generator, reads program templates to control what it generates, and provides a complete, fully-documented *Program Template Language*.

Designed for the IBM PC, XT, AT, PS/2, and compatibles, SoftCode sells for $195 which includes one template set; other template sets are priced at $49. This product requires DOS 2.00 or higher. Reader Service No. 29.
Software Bottling Co.
6600 Long Island Express Wy.
Maspeth, NY 11378
718-458-3700

Opus Systems and **Ibuki** have announced the Lisp Wizard, a Common Lisp co-processor board that allows high performance Lisp-based applications to run on the IBM PC, AT, or compatibles in a pure DOS environment.

Wizard of OS hardware provides virtual memory, multitasking capabilities, 32-bit CPU, on-board FPU, and up to 16 Mbytes of on-board RAM. Discounted prices range from $2,820 (2.5 MIPS, 4 Mbytes) to $8,460 (5 MIPS, 16 Mbytes). Reader Service No. 30.
Opus Systems
20863 Steven Creek Blvd., Bldg. 400
Cupertino, CA 95014
408-446-2210

Recently released by **Applied Logic Systems** is ALS Prolog Macintosh, Version 1.0 compiler, a development environment which broadens the

spectrum of hardware on which ALS Prolog systems are available.

Interaction occurs through text editor windows, and there is a programmer's interface to Quickdraw as well as a graphics window for making pictures with Prolog. Compilation is completely transparent. There is no interpreter and no need to separately link a program once it has been compiled.

The ALS Prolog Macintosh version provides a module system, tail recursion optimization, garbage collection, DCGs, and a debugger. The $349 price includes a programmers' reference manual and one year of updates. Reader Service No. 31.
Applied Logic Systems Inc.
Box 90,
University Station
Syracuse, NY 13210
315-271-3900

Forth Inc. has released chipForth 8051 interactive software for developing embedded systems on the Intel 8051 MPs family. This product includes these features: it is fully interactive (an incircuit emulator is not required); programming can be in assembler, high-level Forth, or both; and a real-time, multitasking executive is included.

It comes with a resident full-screen editor, source program control utilities, online documentation, PROM burning utilities, and source and documentation printing utilities. Also, chipForth uses an IBM PC or compatible as host to write 8051 code and download it to the chip via a serial link.

Available for $3,750, chipForth includes one year of telephone hotline technical support and use of the Forth Inc. Bulletin Board. Reader Service No. 32.
Forth Inc.
111 N Sepulveda Blvd.
Manhattan Beach, CA 90266
213-372-8493
800-55-FORTH

Quantum Software has announced Asmflow, an assembly language flow charting and source code analysis tool for MS-DOS assembly language programmers. Asmflow automatically generates flow charts and call tree diagrams from Microsoft Macro Assembler (MASM) source files.

The call tree features include the ability to control the maximum nesting level and to determine whether duplicate sub-trees are displayed. You can conditionally include macro calls, external references, recursive code, and the stack depth at each procedure call.

Other features include maximum stack size determination, CPU timing analysis, procedural cross-reference, CPU register analysis, and flagging of questionable code. AsmFlow can be used on all IBM PCs and compatibles. The product costs $99.95. Reader Service No. 33.
Quantum Software
19855 Stevens Creek Blvd., Ste. 154
Cupertino, CA 95014
408-244-6826

Recently released by **Mortice Kern Systems** is Version 2.3 of the MKS Toolkit which provides implementations of over 115 Unix System V compatible commands designed to run in the DOS environment on the IBM PS/2 and IBM PC families and compatibles. This item sells for $169.

Also announced is the release of two new software items, the MKS Trilogy which provides Xenix and Microport users with three replacement utilities — the MKS Korn Shell, MKS Awk, and MKS Crypt — and MKS RCS (Revision Control System) that manages multiple revisions of text files and program source code on Xenix and Microport systems. The Trilogy is priced at $119 and the RCS at $189. Reader Service No. 34.
Mortice Kern Systems Inc.
35 King St. N
Waterloo, Ont., Canada N2J2W9
519-884-2251
800-265-2797

CSSL's System Sleuth, a new PC diagnostic software program, is an all-in-one configuration and troubleshooting utility which supports PC/MS-DOS or PC-MOS/386. Starting with simple information like the amount of memory installed and general system configuration, the program progresses lower into the system, providing information such as device chain maps, user memory maps with currently loaded TSRs, and the interrupt chain. Also, hardware information (types and brands of expansion boards installed) is available.

Additionally, System Sleuth will independently verify system configuration information such as absolute disk parameters determined through probing the physical device. Priced at $149, this product can also break out the available I/O expansion card ROM area with a resolution of 2K. Reader Service No. 35.
CSSL Inc.
909 Electric Ave., Ste. 202
Seal Beach, CA 90740
213-493-2471

Hardware

Microcosm Inc. has released hyperICE-386, a real-time in-circuit emulator that allows engineers to develop systems based on the Intel 80386 microprocessor. HyperICE-386 consists of a microprocessor-specific probe and a universal chassis.

OF INTEREST

(continued from page 144)

The product offers software and hardware engineers a host-independent way to debug systems at full *speed* by emulating the 80386 at clock speeds up to 25 MHz. Also, overlay memory runs with zero wait states at speeds up to 20 MHz.

Instead of using the 80386's debug registers, hyperICE-386 incorporates three 8,000-gate arrays. hyperICE-386 provides extensive trigger logic, monitoring all address, data and status lines, and 16 logic probe lines. Triggering controls trace collection and can stop the emulator at any point, down to the register level. Also, the transparent tracing of hyperICE-386 gives users a record of all chip functions as it runs in real time.

Fully configured systems, including chassis and probe, start at $17,500. Reader Service No. 20.

Microcosm Inc.
15275-E SW Koll Pkwy.
Beaverton, OR 97006
503-626-6100

SDI Signal-to-Disk Interface by **Ariel Corp**. allows host independent, real-time, full-bandwidth data acquisition to disk, as well as signal editing and processing for any PC. The new SDI systems are available with 50- or 250-Mbyte internal- or external-hard disks or optical (WORM) disks for archives, permitting hours of storage at maximum bandwidth.

SDI permits direct-to-disk recording/playback of 16-bit data at sample rates of up to 50 KHz on two channels simultaneously. The recorded signal can be viewed graphically and edited; parts of the signal can be marked, zoomed, moved, repeated, attenuated, and deleted; special effects can be added; and the revised signal can be saved and played back. System prices start at $3,495. Reader Service No. 22.

Ariel Corp.
110 Greene St., Ste. 404
New York, NY 10012
212-925-4155

DDJ

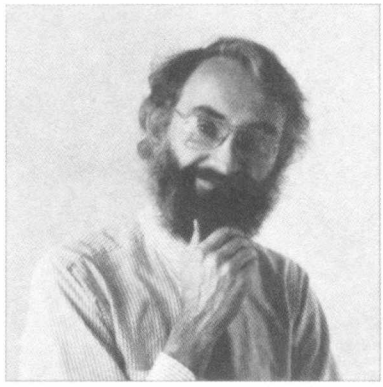
DOUBLE CROSS

Listing One *(Listing continued, text begins on page 46.)*

```
850:                RefPtrs^[i].Last := 0;
851:              END;
852:            j := j + 1;
853:          END;
854:      END;
855:
856:
857:    WriteLn;
858:    WriteLn;
859:
860: END;
861:
862:
863:
864: PROCEDURE WriteResults;
865:
866: BEGIN
867:
868:    WriteLn('Writing output file.');
869:
870:    Page := 1;
871:    LastSymLine := '';
872:    SendCRs := TRUE;
873:    Header(FALSE);
874:
875:    FOR i := 1 TO PublicCount DO
876:      BEGIN
877:        WriteOutCr;
878:        WriteOutSym(PublicVarArr^[i]^+copy(
            separator,1,33-length(PublicVarArr^[i]^)
            )+FileName[PublicFileArr[i]]^);
879:        SameSym := TRUE;
880:        IF RefPtrs^[i].First = 0 THEN
881:          WriteOutCr
882:        ELSE
883:          BEGIN
884:
885:            k := 0;
886:            l := 0;
887:            j := RefPtrs^[i].First;
888:            REPEAT
889:
890:              RANum := ((j-1) Shr Refshift) + 1;
891:              RAOff := ((j-1) AND (Refsize-1)) + 1;
892:              m := RefArrs[RANum]^[RAOff].Filenum;
893:              junk := FileName[RefArrs[RANum]^
                          [RAOff].Filenum]^;
894:
895:              IF k = m THEN
896:                BEGIN
897:                  IF (l >= 8) THEN
898:                  BEGIN
899:                    WriteOutCr;
900:                    WriteOut(copy(blanks,1,18));
901:                    l := 0;
902:                  END;
903:                END
904:              ELSE
905:                BEGIN
906:                  l := 0;
907:                  WriteOutCr;
908:                  WriteOut('     ');
909:                  WriteOut(junk+copy(
                              blanks,1,13-length(junk)));
910:                END;
911:
912:              Str(RefArrs[RANum]^[RAOff].Linenum,Temp);
913:              Temp := copy(blanks,1,6-length(temp))+temp;
914:              WriteOut(temp);
915:              IF RefArrs[RANum]^[RAOff].Define THEN
916:                WriteOut('*')
917:              ELSE
918:                WriteOut(' ');
919:              l := l + 1;
920:              k := m;
921:              j := RefArrs[RANum]^[RAOff].Next;
922:            UNTIL j = 0;
923:            SameSym := FALSE;
924:            WriteOutCr;
925:            WriteOutCr;
926:          END;
927:      END;
928:
929:    WriteOutCr;
930:    Str(PublicCount,Temp);
931:    WriteOutLn('Number of publics: '+Temp);
932:    Str(Refcount,Temp);
933:    WriteOutLn('Number of references: '+Temp);
934:    WriteOut(chr(12));
935:
936: END;
937:
938:
939:
940: PROCEDURE Terminate;
941:
942: BEGIN
943:
944:    Close(infile);
945:    IOCheck(infilename);
946:    Close(outfile);
947:    IOCheck(outfilename);
948:
949:    WriteLn('Done.');
950:    Release(Topofheap);
951:
952: END;
953:
954:
955:
956: BEGIN
957:
958:    Initialize;
959:
960:    Pass1;
961:
962:    Pass2;
963:
964:    CompactResults;
965:
966:    WriteLn('Sorting');
967:
968:    Megasort(PublicVarArr^,PublicFileArr,
                  RefPtrs^,32,PublicCount);
969:
970:    WriteResults;
971:
972:    Terminate;
973:
974: END.
```

End Listing

SWAINE'S FLAMES



I was up to my ears in galleys for my book on Hypertalk;
I had barely time to Paradigm and no time left to squawk
(or Flame or rage) on the final page, as it is my wont to do.
So I sent a request
for a Flaming guest
and Erickson came through.
— Michael Swaine

What with enrollments down, costs up, and budgets tight, it comes as no surprise that university-level computer-science programs are facing a multitude of challenges. As a case in point, consider what's currently going on at the University of California's Berkeley campus, a school with a computer-science department that deservedly has the reputation as one of the finest in the country.

Both faculty and students alike agree that the CS building and facilities are overcrowded and, relatively speaking, antiquated. With this in mind, the department began a couple of years ago to lobby for a new building and, after months of stalling, the university agreed in principle that a new building is indeed in order. That's the good news. The bad news is that the university didn't offer to set aside any land on which to construct the building, nor did the university agree to provide money to pay for it. It was only after more intense lobbying on the part of the CS faculty that the university finally allotted some land for construction. (It should be mentioned that suitable building sites are extremely scarce on campus, as they are throughout the City of Berkeley.)

That's the good news. The bad news is that the land for the new CS building just happens to be on top of a nuclear reactor that is in the slow process of being shut down. Still, engineers said that it is possible to put a structure on top of the reactor by erecting the building on stilts around the reactor's perimeter. Not the ideal solution, but certainly better than no chance of a building at all, and the CS faculty was happy to have some commitment.

That's the good news. The bad news is that before construction can begin, the nuclear fuel rods, radioactive shielding, and other associated wastes are still in the reactor and need to be removed. In most instances, there's nothing particularly difficult about dismantling a nuclear reactor; it's been done many times without mishap. The problem in this case, however, is that Berkeley has been declared a nuclear-free zone, meaning it is against the law to transport such materials over the city streets. The only saving grace is that the reactor is licensed by the Nuclear Regulatory Commission, which operates under federal laws that supercede local statutes, and so the radioactive material can be trucked out over the objections of the City of Berkeley. Protest leaders have already promised to disrupt the process, and there's little doubt that demonstrations will occur.

Keep in mind that all of these problems concern only the building site itself. The other side of the coin involves the money needed for construction. Because the university didn't provide any money for construction, it is up to the CS faculty to raise, from private sources, the $30 million necessary to put up the building. So far, they've managed to scrape up nearly $5 million from private sources that include Lockheed, several Japanese computer firms, and UC Berkeley alumnus Steve Wozniak.

As strange as it may seem, major computer companies in this country have not made any solid commitments even though they will eventually stand to benefit from the rich pool of talent that will graduate from Berkeley. Granted, it is hard to explain to stockholders why paying for someone else's bricks and mortar is necessary, especially when profits may be down. And, although no one will go on record as saying so, the current confusion in the Unix community seems to be playing a role too. One faculty member claimed that IBM and other backers of the Open Systems Foundation don't necessarily want to lend a hand to UC Berkeley, an institution that is perceived to be a supporter of the Unix standard championed by AT&T and Sun Microsystems, and a possible competitor to the proposed IBM-backed OSF standard.

Richard Fateman, chairman of UC Berkeley's CS department, says that he is tired of fund-raising and would rather be devoting his time to research and teaching. Other faculty members expressed similar sentiments. It seems to me that the problem with this entire situation, which no doubt is being repeated to some degree in universities across the country, is that teachers are supposed to teach, yet they can't be teaching if they have to go out and find money to build their own buildings. No one wins in a situation like this—the country, the computer industry, and certainly not the students.

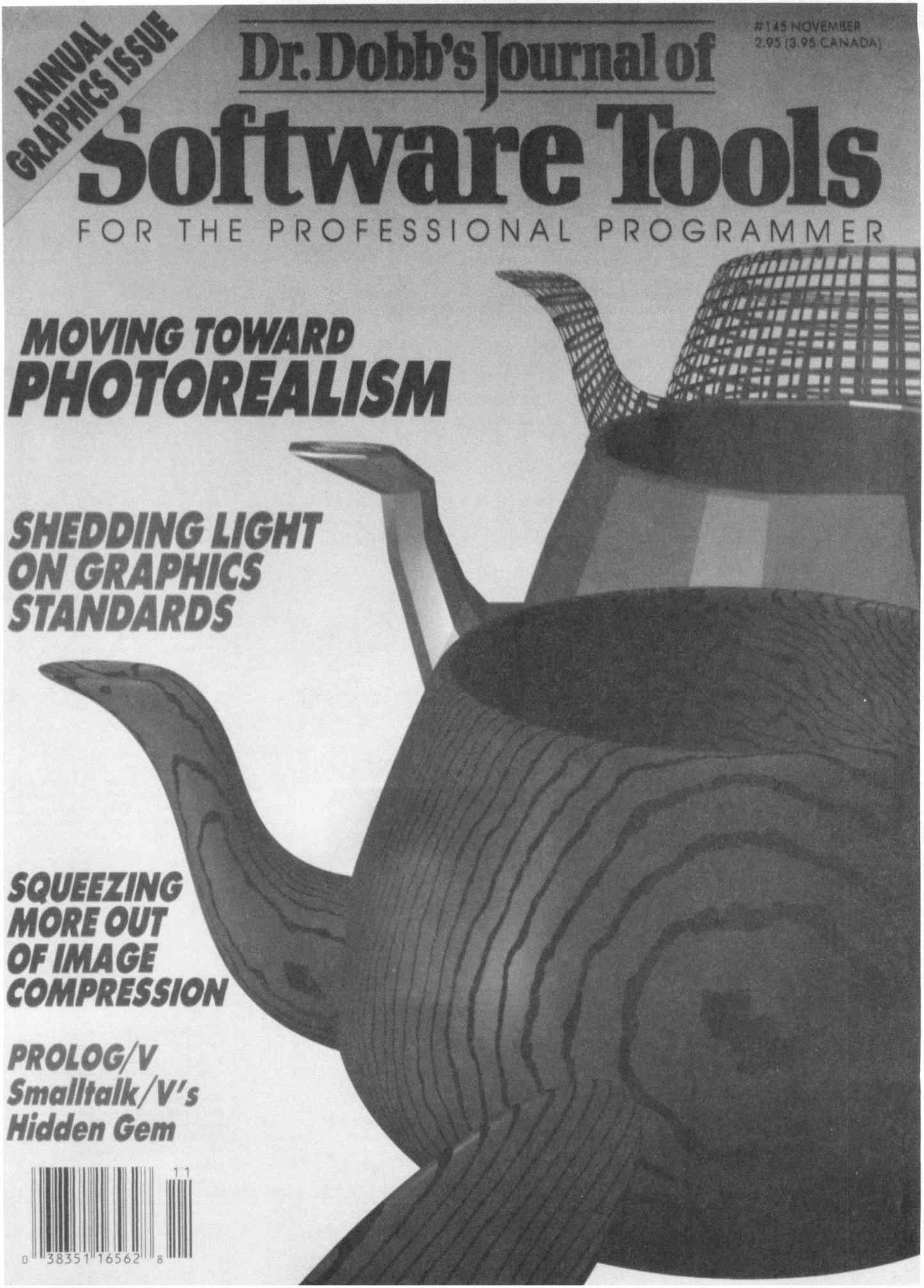
Jonathan Erickson
editor-in-chief

#145 NOVEMBER
2.95 (3.95 CANADA)

ANNUAL GRAPHICS ISSUE

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER



MOVING TOWARD PHOTOREALISM

SHEDDING LIGHT ON GRAPHICS STANDARDS

SQUEEZING MORE OUT OF IMAGE COMPRESSION

PROLOG/V Smalltalk/V's Hidden Gem

0 38351 16562 8 11

ARTICLES

Photorealism in Computer Graphics 18
by Steve Upstill
Is it real or is it Pixar? Steve describes how advancements in shading and light refraction bring computer imaging even closer to the real thing.

Graphics Standards ▶ **Putting Graphical Interfaces into Perspective** 32
by Kent Dahlgren
It's getting so you need a *program* just to know the players in what's turning into a "battle of the graphics standards."

Graphics Coprocessors ▶ **Image Compression via Compilation** 42
by Victor Duvanenko
Manipulating the bulk of data required by graphic images eats up time and disk space —a graphics coprocessor can save you both.

C ▶ **Dynamic Run-Time Structures** 50
by Todd King
Using dynamically defined data structures is one way to create flexible, yet powerful applications.

Mapping DOS Memory Allocations 56
by Robert J. Moore
Avoid a world of pain, understand how DOS uses memory control blocks.

Basic and Assembly Language ▶ **Inserting Elements into a Basic Integer Array** 66
by Bruce Tonkin
Mixing assembly language with Basic yields some stunning performance improvements. And, as Bruce explains, it's not even that hard to do.

REVIEWS

Smalltalk and Prolog ▶ **EXAMINING ROOM** 68
coordinated by Ron Copeland
Products examined from the programmer's perspective. This month, Gregory Lazarev takes a look at Prolog/V, a hidden gem included *free* with Smalltalk/V.

COLUMNS

C PROGRAMMING 104
by Al Stevens
Al's project continues to grow with a general purpose text editor that (somehow) leads to a discussion of maintaining comments in code.

STRUCTURED PROGRAMMING 110
by Kent Porter
Kent takes a first look at Borland's Turbo Pascal 5.0 with the long-awaited integrated debugger.

PROGRAMMING PARADIGMS 116
by Michael Swaine
Michael ruminates over alternatives to conventional programming languages and wonders how (if) they'll affect our future.

FORUM

EDITORIAL 6
by Jonathan Erickson

RUNNING LIGHT 8
by Ron Copeland

LETTERS 12
by you

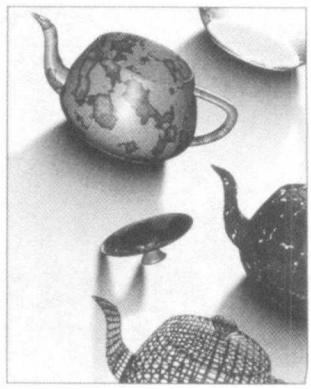
SWAINE'S FLAMES 144
by Jonathan Erickson

PROGRAMMER'S SERVICES

ADVERTISER INDEX 128
where to go for more information on products

OF INTEREST 132
brief product descriptions

PROGRAMMER'S MARKETPLACE 140
classified ads



The history of computer graphics — from wireframe, through polygons, and on to photorealistic curved surfaces —is summed up on this month's cover and feature art. The teapots were rendered at Pixar by Steve Upstill using a hardware accelerator and software driven by Pixar's RenderMan™ Interface. Thanks to Jeff Mock for the shader used to create the wireframe teapot and to Mark Leather for the wood grain shader. In the accompanying article, Darwyn Peachey wrote the shader for the marble texture, while Bob Drebin wrote the shader for the shadows.

Next Issue

Operating systems is our focus in December when we'll take a look at the major players, including DOS, Unix, and OS/2, and we'll examine how you can design software to run on multiple operating system platforms.

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, except semimonthly in June and October by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 3771, Escondido, CA 92025. **ISSN 0888-3076**

EDITORIAL

If a single word sums up the current state of computing, that word is confusion. It's hard to remember a time when the PC industry was in a more confounding state of affairs. Now, more than ever, there is a bewildering array of choices in everything from operating systems to hardware architectures. And the choices aren't getting any easier.

Take a look at operating systems, for example. It was only a few years ago that developers had a scarcity of options. There was CP/M, AppleDOS, and a few other systems available —TRS-DOS (boy, there's a stab from the past) was one I worked with —but for the most part, choices were few and the stakes relatively low.

Today, developers must decide between DOS, OS/2, Unix (pick your favorite flavor), the Macintosh, and a host of other more specialized alternatives. If you pick the wrong development platform, the results may not be a pretty sight.

In the same sense, the Macintosh was basically the only dominant windowing system three or four years ago. Today, there's at least a dozen, including Windows/Presentation Manager, the Macintosh, X-Windows, Rooms, GEM, and New Wave, to mention a few.

Now, throw in a wildcard, like Display Postscript (Steve Jobs' choice), and take into consideration the rumor that former operating system rivals Digital Research and Microsoft are considering offering DRI's GEM application-development tool kit as a development tool for Presentation Manager. Now you're faced with a perplexing array of opportunities or quagmires, depending on your perspective.

And things are just as confusing on the hardware side. We've somehow moved from a couple of accepted architectures (the AT bus and a closed-system Mac) to a plethora of standards that include the Micro Channel Architecture (MCA), the NuBus, and, more recently, the Extended Industry Standard Architecture (EISA). (EISA is a 32-bit extension of the AT bus that has been endorsed by several PC manufacturers —IBM not among them —that supposedly provides the performance benefits of the MCA, yet is compatible with existing AT-bus cards.) Not to mention, last month's introduction of Steve Jobs' NeXT workstation and the whisperings about an 80386 machine waiting in the wings at Apple. I'm getting an ulcer just writing about it.

Now consider the various incarnations and implementations of programming languages. Should you go with C or begin thinking about C++? What about Modula-2? There are some powerful new Modula compilers out there that deserve consideration, and the renewed promise of Basic (hearken back to Bill Gates' remarks about Object-Basic) as a serious development platform.

Anecdotically, a developer I know wrote a complicated Windows application in Pascal. He wound up rewriting the entire project after he figured out that Modula-2 would enable him to better accomplish what he was trying to do.

Regardless of what the soothsayers predicted a couple of years ago, nothing about computing —from either the developer's or the end user's perspective —is getting any easier. Now, I'm not complaining about more powerful systems, you understand, or about the diversity of choices we're faced with today. I'll put it this way, it's better to be rich and healthy than sick and poor.

The point to all of this is that developing for and porting between different environments is no easy matter. Developers must choose wisely and well. Some platforms won't be alive and kicking tomorrow. For that matter, neither will developers who've made the wrong choice.

Jonathan Erickson
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Jonathan Erickson*
Managing Editor *Monica E. Berg*
Senior Technical Editor *Kent Porter*
Senior Editor *Ron Copeland*
Editorial Assistant *Kathleen Evans Ralston*
Technical Editor *Richard Relph*
Contributing Editors *Al Stevens, Martin Tracy, David Betz, Tom Genereaux*
Copy Editors *Rhoda Simmons, Kevin Shafer*
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Lisa Schneider*
Technical Illustrator *Lynn Sanford*
Typographers *Lorraine Buckland, Margaret Anderson*
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Direct Marketing Manager *Andrea Weingart*
Direct Marketing Coord. *Francesca Davies*
Newsstand Manager *Sarah Frisbie*
Fulfullment Coord. *Joan Saspo*

Administration
Vice President of Finance and Operations *Kate Wheat*
Business Manager *Betty Arsene*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Marketing Assistant *Sara Noah Ruddy*
Account Managers *see page 145*

Publisher
Peter Hutchinson

Article Submissions: Send manuscripts and disk (with article and listings) to the associate editor.

***DDJ* on CompuServe:** Type GO DDJ

Subscription: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds with additional postage. Add $13 per year for surface mail to all countries or add $33 per year for airmail to Canada and Mexico; add $32 per year for airlift/surface to all other countries.

Customer Service: For subscription orders and changes of address call toll-free 800-354-8400 or write: *Dr. Dobb's Journal* subscription Dept., P.O. Box 3771, Escondido, CA 92025. For all other subscriber inquiries call 800-321-3333 (in California 800-331-4164, outside the U.S. 619-485-9623). For book/software orders call 800-533-4372 (in California 800-356-2002).

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

ABC

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Fosbay*
Vice President of Publishing *William P. Howard*

RUNNING LIGHT

I believe that the most successful programmers in the next decade will be those who carry in their toolkits, among their shiny metric and non-metric algorithms, a rich set of paradigms.

Mike Swaine
—DDJ *editor-who-is-elsewhere*

The best software is software that is hand-crafted by artisan programmers —*DDJ's* bread and butter. Handcrafted has always been a synonym for the best in any category. It has also been true that, historically, while an artisan class (in any trade) was being loudly praised and greatly revered, it was also being systematically eliminated in favor of some type of mass production technology.

While it may be unpopular to say so, one man, one vision continues to be the best way to develop the best software. Unfortunately, in the main, neither economic necessity nor the economies of scale are apt to show any mercy in the continuing battle of quality over quantity. Ways of getting more from fewer are likely to remain a survival-oriented company's dominant goal. But while competitive pressure makes strong demands, fortunately, consumers will always favor the best.

What's obviously needed is some way to cater to economic reality while getting the best from the best. Everybody wins. Sounds good, but how?

Approaches to writing software have always embodied the balance between flexibility and complexity on the one hand, and rigid but focused simplicity on the other. It's either hard and powerful, or easy and simple. On the easy side high-level languages now offer elaborate object-oriented systems that can automatically generate applications from examples or from iconic connections. The down side is that by focusing programming power in certain directions high-level languages limit, in differing degrees, the number of directions that can be chosen —reducing the freedom to excel, to innovate.

On the flip side, languages based on the linguistic approach have the discreet expressive power to precisely thread the eye of a needle without touching the sides. But in a large programming project under stringent deadlines, this kind of power, coupled with unlimited creativity, can easily become a logistical nightmare.

In the past, procedural languages have served programmers well. There is ample evidence to support this. The bulk of today's software is visually enticing and heartily robust, with functions steadfastly following form. We must make sure that this remains the case.

The prototyping pundit himself, Ted Lewis of Oregon State University, differentiates between these two approaches by reducing them to the nature of their approach to a problem. He describes these approaches as "telling" versus "showing."

Simply, Ted defines telling as issuing a discreet set of instructions. This involves two symbolic transductions: from the idea to its componential representations, then back again. Showing he defines as manipulating 'objects' directly. Showing only involves a single translation —from the idea directly to its implementation. Showing is usually regarded by learning and communications theorists as the least dissonance-inducing form of communication.

Right now, showing a computer what to do is a less evolved technology than telling it what to do via a programming language. Its promise is largely unfulfilled. But it's clear that there are real advantages to be gained from object-oriented programming in languages like Smalltalk, C++, and Object-Pascal. (Even Philippe Kahn seems to think so, or so he hinted at a recent press conference where he talked about products forthcoming from Borland in 1989.) In a more applied sense, similar advantages are gained from graphics specification/manipulation tools such as HOOPS and RenderMan.

Indeed, perhaps the ideal approach is a model that supports and embodies the desirable features of both, either from within a single programming environment or by supporting a multiple compiler integrated/integrating environment. Both Borland and Microsoft have indicated ongoing commitments to supplying a multiple language environment at some point in the foreseeable future, as well as indicating their strategic commitment to object-oriented language development. Time will tell.

We live in interesting times. (This is a Chinese curse, by the way.) Right now we're caught smack in the middle of the paradigmic shift. Not a comfortable place to be, but then again, it's certainly not boring.

Ron Copeland
senior editor

ARCHIVES

Ten Years ago in *DDJ*

"Most manufacturers suffer from advanced, chronic bandwagonism: every time any one of them comes up with something new, it is immediately copied a dozen times over. This bandwagonism hurts everyone: the innovators have their ideas stolen; the bandwagonists must compete against many other companies making an identical product, and the users get little freedom of choice in buying their computers and accessories. Bandwagonism has also tended to establish standards, most of them informal, incomplete, and harmful." —*David Chapman "Programming Languages and Standards,"* DDJ, *November 1978.*

Yet It Was Quality Time . . .

"Typical computer resources of many high schools consist of a single port connected to a district-owned BASIC timesharing computer or to a state or university consortium. Since only one student at a time may use the computer, each student in a class of 30 receives under nine minutes of time per week." —*Jeff Levinsky, "Chaos: An Interactive Timeshared Operating System for the 8080,"* DDJ, *January 1979.*

But Could He Program in Assembly Language?

"Bing Crosby recognized that the microphone opened up the possibility for more intimate communication with an audience . . . and developed new techniques to go with the new medium. As computers get smaller and are used in a more personal way by more people, we have to develop techniques of making software that can reduce the psychological distance between computers and their users." —*Paul Heckel, "Zoomracks: Designing a New Software Metaphor,"* DDJ, *November 1985.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Algebraic Specification is the Answer

Dear *DDJ*,

In Michael Swaine's recent column about object-oriented programming on SD '88 ("Programming Paradigms," June issue) there was a question: Are there any rules to decide what objects to define? I think —in contrast to the panelists —there is a good way to construct programs with well-chosen objects. Of course, if you start from scratch nothing will be changed for the better.

My answer is simply to use algebraic specification to define the problem and its solution. That means use a specific style of functional programming with precise semantics. You have to define entities by functions and conditional equations to enhance and combine the specifications to describe the problem at hand. This way you will easily get a well-structured problem solution, telling *what* is to be done.

And now consider object-oriented programming as the way to tell *how* the solution is to be realized on the computer. You will find a direct correspondence between specification modules and objects needed in the implementation. I can't speak here about all the other —and more essential —merits of this approach because there is not enough room. But I think we should favor the use of formal methods already in the design phase to overcome such questions as the one mentioned above —and a lot of other difficulties in programming.

Dr. Lutz Pietschker
Muhldorf, West Germany

Source Code Correction

Dear *DDJ*,

I like Kent Porter's "sub"stitute for the DOS *dir* command very much ("Structured Programming," August issue). You may be interested in a minor error in the source code: in function *SizeInK*, "IF size MOD Blocksize < > 0" should be "IF *files*.size MOD Blocksize < > 0." I stumbled on this when I used "sub" on a TEX pixel directory which happened to contain files with exactly 4,096 bytes.

William F. Trench
Trinity University
San Antonio, Texas

Reader Offers Split Screen - Action Diagram Editor

Dear *DDJ*,

It was interesting to read Martin Stitt's article on action diagrams —or action charts as he calls them ("Using Action Charts," September issue). These diagrams have been around for some time, competing against other diagramming methods for attention. I find action diagrams rather useful for defining mini specs in data flow diagrams. They are clear, concise, and much more accepted by the user community and the programming staff than structured English.

However, typing all those macros to create such a diagram is tedious and diminishes its effectiveness. I have created a split screen action diagram editor which relieves analysts of having to worry about the constructs, and instead, they can concentrate on showing the logic flow. I urge readers to contact me for a free copy at 6087 Dana Dr., Norcross, GA 30093; 404-925-3122.

Sam Johnson
Norcross, Ga.

Real Programmers Don't Use Assembly Language

Dear *DDJ*,

I really appreciated Ratcliff's and Metzener's article on Pattern Matching ("Pattern Matching by Gestalt," July issue). It was excellent! I was also quite intrigued with the algorithm itself, but like all *real* programmers, I dislike having to use assembly language.

I certainly don't believe that assembly language is necessary in this case. I therefore implemented it in C using a recursive function. Anyone interested can have it (Listing Two, page 118).

The execution times of the code on a 6-MHz AT in SCO Xenix C were about twice that of what were reported in the article on an 8-MHz machine. I believe the trade-off of time for readability is quite worthwhile.

Thanks again for a great article.

Joe Preston
Agent Systems Inc.
Dallas, Texas

Dear *DDJ*,

I really enjoyed Ratcliff's and Metzner's article "Pattern Matching by Gestalt" in the July issue. For your readers who use computers that are not IBM PC compatible, I have rewritten the *simil* and *compare* routines in C (Listing One, page 118). I have left the variable names and program structure very close to the assembler version to make it easy to follow.

Rick Orsborn
Wheat Ridge, Colo.

Debuggers' Debilities

Dear *DDJ*,

The review of the current state of C compilers in the August issue was very interesting. ("Speed Trials: Five Cs Compared.") I was especially pleased to see that you tested code generation and its correctness. This gives a better feeling for the quality of the compiler than using only the benchmark timings.

However, I was puzzled by the emphasis on debuggers. There were no details on what made one debugger stand above the others, but it's something of a moot point anyway. Software debuggers are not the best tool for the job and should not influence the rating of a compiler.

Specifically, every software debugger I've seen is weak in three places. I've use Microsoft's Codeview, so I'll use it to illustrate my points, but the same criticisms apply to other debuggers too.

1. Transparency. Adding the debugger to the system should not change the behavior of the application program. Software debuggers almost all fail here because they load below the program in RAM. A program that passes data on the stack (like C, for example,) may act differently when loaded at the higher location. Beside that, the RAM used by the debugger is that much less heap available to the program. Codeview is particularly bad here —not many serious programs won't notice the loss of 200K.
2. Independence. The performance of the debugger should require as little as possible of the other parts of the system. At a minimum, it should not expect the program to leave the interrupt vector table alone. Good ones shouldn't depend on DOS, either. Codeview fails here, too. Trash the keyboard interrupt vector, and down it goes.
3. Reliability. The debugger should be

continued on page 16

LETTERS
(continued from page 12)

able to protect itself from the application. C programs are famous for crashing when people use uninitialized pointers and romp through memory that doesn't belong to them. Some software debuggers claim to use the capabilities of the 80386 to protect themselves. Some may work. The one I tried didn't.

The only thing I've found that meets these requirements is a hardware-based debugger. I use Periscope, but there are others out there too. Periscope uses 64K of memory on an expansion board to store its code and data. The memory can be installed in memory space, which is unavailable to the program in the first place, so you lose no bytes for your application. The memory is write-protected so your application can't trash it, too. Furthermore, Periscope reloads the interrupt vectors it needs for I/O while it is active, so your programs can even trash them and not crash the debugger. Their real trump, though, is the hardware trace circuitry on the board. You can run at full speed, quietly monitoring the running application; then break out to the debugger when a programmable break condition occurs; and then examine the hardware traceback buffer to see exactly what your program was doing before and after the break condition occurred. Periscope even has source-level debugging if that's what you want. I've used it to debug a 500K application which loads under DOS then deletes DOS and disk BIOS from the system entirely while it's running to use the extra heap. Try *that* with Codeview!

In short, software debuggers fall far short of what is attainable with hardware based products. This being the case, it's not fair to rate these top line compilers on the basis of the accompanying debuggers. Anyone who's serious about debugging (meaning their job is on the line) probably won't use them anyway. The protection capabilities of the 80386 open the way for software products that make the grade, but they aren't here yet, and you can bet they won't be freebies when they come.

Jim Castleberry
Orlando, Fla.

DDJ

Ezra's personal touch in "graphic software" catches the attention of the local constabulary.

Photorealism in Computer Graphics

by Steve Upstill

With the phenomenal increase in computer power of recent years, computer-graphic imagery (CGI) has emerged as one of the easiest ways to consume excess cycles. For a long time the novelty of creating pictures on a computer was enough to justify the effort. Like Johnson's dog, who walked on its hind legs, "It is not done well; but one is surprised to find it done at all."

But expectations for the quality of CGI have increased. With viewers jaded by a barrage of imagery in television commercials and network logos, that computer look grows old. The ambitions of computer-graphic researchers, as published in SIGGRAPH and other forums, have risen proportionately, with the current goal being to produce synthetic images of realism that are comparable to photographs.

Figure 1, page 20, which depicts a transparent light bulb, shows an example of the kind of complexity required by photorealistic computer graphics. The light bulb includes a variety of light sources and surface types, previously mapped imagery, transparency, reflections, a glow, and so on. It also includes small details, like the roughness of the surface of the glass, which escape conscious notice but nonetheless lend important depth to its realism.

A quick glance around any natural environment reveals a mind-boggling variety of textures, reflections, diffusions, and optical "special effects;" the kinds of phenomena nature throws at us willy-nilly. Ask a random graphics hacker to reproduce such a scene as a digital image, and the reaction would likely be a dropped jaw, a derisive snort, or impassioned rationalization. The least likely response would be "Sure, right away."

The Traditional Approach

There is a fundamental flaw in the way programmers have approached the problem of realism and computer graphics; the flaw is typical of the way science proceeds. Starting with the crudest imaginable imagery (such as wire-frame drawings and pen plots), rendering methods have improved over the years so that they now can provide flat polygonal surfaces, and shading techniques have been developed to provide correct metallic surfaces and texture mapping that do indeed make pictures look better. As they have appeared, these methods have been incorporated into rendering and shading programs, which have tended to grow in size and complexity.

As software engineering this approach is fundamentally flawed, but computer graphics has nonetheless embraced it implicitly. A typical rendering system simulates optical reality using a monolithic program controlled by a selection of options, parameters, and maps of various kinds. New techniques are introduced by hacking the initial implementation, until the program gets so complex that no one can deal with it productively. It is reimplemented from scratch, and the whole cycle begins anew. Even if this cycle could be sustained indefinitely, it would still fall short of its goal, because physical reality is much too complex ever to be encompassed by a monolithic rendering system.

A Different Approach

The problem with traditional approaches to realism and computer graphics is the assumption that all the functionality that will ever be needed can be packed into a single program and accessed by manipulating options, flags, and parameters. But physical reality in all its richness cannot be reproduced by twiddling knobs.

The idea of the RenderMan Shading Language, (see sidebar) on the other hand, is that a rendering system needs to be extensible and should provide the programmer with a solid basic system to which new tools can be added as the need arises. Not only will the programmer be able to customize the system to meet unanticipated needs, but libraries

Steve Upstill is a graphics researcher for Pixar and can be reached at 3240 Kerner Blvd., San Rafael, CA 94901.

of tools, created by third-party developers independent of both the end user and the basic system, can be developed.

In short, the shading language is a door to extensibility in rendering. Using the shading language a programmer writes a small procedure for performing one of several functions in the rendering process. For example, a light source is defined as a shader whose function is to calculate the light striking a surface with a particular orientation at a particular point. A surface shader has the task, given the impinging light, of returning the light reflected from the surface toward the eye.

Once defined, a shader is embedded in a pre-existing rendering system which takes care of performing geometric transformations, calculating hidden surfaces, providing a shader with its parameters and calling them, and more. A shader has very tightly defined tasks, and so it is typically only a few lines of code in length. It is very much the tail wagging the dog. The key to the usefulness of the shading language is finding the right set of tasks for a shader to perform, making the task as simple and orthogonal as possible.

Design

Fundamental to the design of the shading language is the distinction between shape and shading. Shape concerns the geometry of a scene to be rendered: the surfaces used to describe the objects in the scene, their placement in the world, the positioning of a camera and light sources to illuminate them, and so on. The geometry of the scene is used to project the image data onto an image plane and determine which surfaces are visible from a given viewpoint.

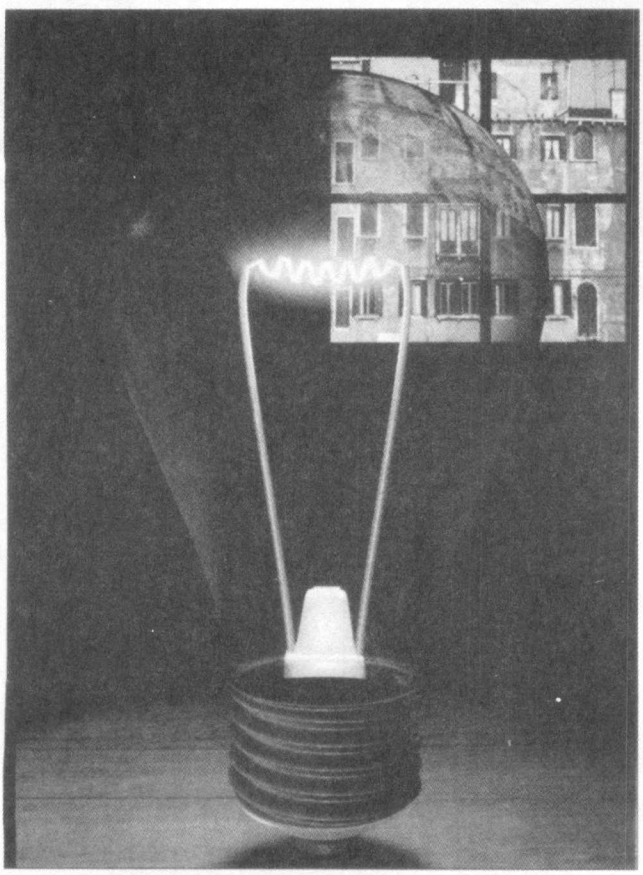


Figure 1: *A transparent light bulb illustrates the complexity required by photorealistic computer graphics*

Shading, on the other hand, concerns the appearance of the objects in the scene, how they look. What color is that wall? Is that telephone shiny or matte? Is that real wood grain or synthetic? Is the material on that chair cloth or leather or vinyl?

Dealing with shape in the context of rendering is a fairly well understood process. Projective geometry and hidden-surface removal can be sealed into a black box with a minimum of interference from users. It makes sense to do so, because hidden-surface removal is both troublesome to implement and uninteresting to users. Shading, by contrast, demands a maximum of accessibility, because it exerts a powerful influence over the quality of imagery, it can be made fairly straightforward, and it is interesting to users.

Most of the visual interest in the world, the detail and small variations that give an image a sense of reality, can be expressed in shading. The crude shape of a chair does not make it seem real; the play of light on its fabric and the highlights on the wood finish, among other factors, do.

Modularity

The second important facet of the shading language is to break the shading process into functionally independent units, to simplify each one and allow shaders of different kinds to be mixed and matched without mutual interference. Each of these units is implemented as a procedure in the shading language, termed a *shader.*

Figure 2, next page, shows a model of the shading process in terms of shaders. Each block shows a type of shader. The only interaction between one module and another is the data items (light color, primarily) that move between modules. From the standpoint of the model as a whole, the internal operation of a module is completely unconstrained. It doesn't matter how the surface module arrives at its reflected color; the only important thing is that it produce a reflected color. You might think of shaders as "white boxes." The normal view of programming procedures is that of fixed "black boxes," which have well-defined input and output and unknown internals, so that they can be assembled into large systems based only on that external view. A shader, by contrast, is fit into a predefined system (the renderer), and the user's freedom lies in implementing the module itself.

Shader Types

The RenderMan Interface specifies eight types of shaders. The types of shader are distinguished by purpose, the kinds of output they produce, and what inputs they use. The first five are concerned with what might traditionally be called shading, and the examples later in this article are chosen from these. The remaining three, not discussed any further here, are applications of shaders to other parts of the picture production process.

Light Source —A light source shader is given the position of a point on a surface and a direction vector. It returns the color and opacity of the light striking that point from that direction. Unlike most other shaders, there may be several light sources extant at one time. They are maintained in a list, and any set of them can be turned on or off for any surface in a scene.

Volume —A volume shader is associated with an object, a closed collection of surfaces. As light passes through the object, an **interior volume shader** is called to modify the light, presumably according to the material making up the object. When the light emerges from the object, an **exterior volume shader** is called to make any changes to the light in the immediate vicinity of the object.

Displacement —The purpose of a displacement shader is to provide detail over the surface of an object, small variations which would be difficult to specify geometrically. A displacement shader takes the position of a point on a surface and the normal to the surface, warping the surface by moving the point.

Surface —When light strikes a surface, it bounces around in the material of the surface and emerges in some direction ususally with a different color than it had when it arrived. A **surface shader** has the job of computing reflected light in a particular direction (usually, but not always, toward the eye), given a point on the surface, the orientation of the surface, and a set of light sources.

Atmosphere —In moving from a surface toward the camera, light moves through an atmosphere (fog, for example), which can introduce other artifacts. An **atmosphere shader** implements those changes by taking a color and direction of light moving from one point to another, modifying it if necessary.

The three shaders mentioned next are applications of the shading language to processes which, strictly speaking, don't concern shading.

Deformation —The geometric transformation capabilities of the RenderMan Interface include nonlinear transformations of geometric coordinates. These are specified with a **deformation shader**, which takes a position on a surface and outputs a new position.

Projection —Normally a renderer projects the geometry of a scene onto an image plane using a straightforward orthogonal or perspective projection. Other projections can be specified using a **projection shader**, which takes a position in world space and outputs a position in screen space.

Imager —The mapping from floating-point color values to the output values of a pixel can be controlled by an imager shader, which takes an RGB triple on input and places three values, of arbitrary meaning, on output. An imager shader might be used, for example, to convert RGB output into a device-dependent format, such as NTSC or four-color separation.

Characteristics of a Shading Language

The development and runtime environments of the shading language make writing shaders much easier than one might think. A variety of tools (provided by the language) and services (provided by the renderer) often reduce the size of a shader to a few lines of code.

Rendering Environment —When a shader is called, all the work of receiving and transforming geometric data and calculating hidden surfaces is already done. A shader has only a single, tightly constrained role to play.

Precalculated Geometry —The geometric information used by a shader is provided as global variables, which are preset every time the shader is called.

Special Data Types —The shading language provides two special data types, *point* and *color*, for manipulating geometric and shading information. The language includes special operators for *point* data, and the standard programming language operators are overloaded to operate on both types.

Varying *Variables* —A common procedure in surface shading is to bind a set of values to the vertices of a surface (a polygon, say) and interpolate the values across the surface. The same capability is provided in the shading language, but that fact is irrelevant to the shader itself. The interpolation takes place behind the scenes, and a surface shader operates the same whether a variable is bound to the surface as a whole or varies from one point to another.

Integration Constructs —Besides the usual programming language control constructs, such as **for** and **if-then-else**, the RenderMan Shading Language supports three special constructs that make it simple to control spatial integration of light emerging from light sources and impinging on a surface.

Filtered Map Access —Texture maps and other forms of two-dimensional, multi-channel data can be accessed from shaders as simple function calls, with prefiltered return values. Any multichannel value can be stored in a map and then accessed by a shader to control any aspect of shading.

Function Library —The shading language provides an extensive library of math, color, optical, geometric, and noise functions. The common shading functions for calculating ambient, diffuse, specular, and Phong reflections are also predefined.

A Typical Shader

The RenderMan Shading Language will be familiar to anyone experienced with an algebraic programming language, particularly C. It's interesting mostly by contrast, so the best way to present it is to dissect a typical shader.

Example 1, page 22, shows *marble( )*, a surface shader. This shader is called whenever a programmer needs the color of the light reflecting from a surface at a given point. This shader can be roughly divided into two sections. The first section, ending with the *surfcolor* = . . . statement, computes the reflectivity of the surface by using noise to perturb the color of the surface. The second part invokes all relevant light sources to determine the amount of light striking the surface from each, summing the resulting eyeward reflections. The result can be seen applied to the light bulb in Figure 3, page 22.

The first distinction between the shading language and C is that *marble( )* is not a function or a procedure, but a **shader:** that fact is denoted by the keyword **surface** preced-

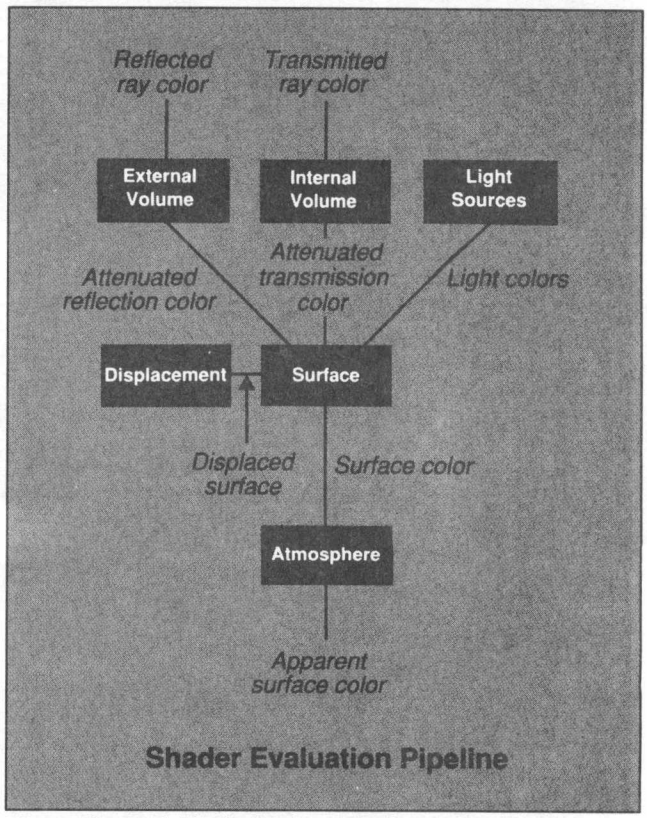


Figure 2: *Each block represents both a subtask of the shading process and a shader used to implement that subtask*

ing the declaration and indicating that this shader is devoted to computing surface reflections. Shaderhood means that the function can be invoked by a program running the RenderMan Interface and linked into the rendering process at runtime.

```
surface
marble( float Kd=.5, Ka=.1;
        color veincolor=0)
{
    float khi ;
    point fnormal ;
    color surfcolor ;
    point freq = 4 * P;
    float turbulence = 0;
    float amplitude = 1, octave;

    /* Make sure the eye-vector points the right way */
    fnormal = faceforward( normalize(N), I ) ;

    /* Accumulate a 3-dimensional noise function over */
    /* 6 octaves, weighting each by 1/f.              */
    for(octave = 0; octave < 6; octave = octave + 1 ) {
        turbulence = turbulence +
            amplitude * abs(.5 - noise(freq));
        amplitude = amplitude * .5;
        freq = freq * 2;
    }

    turbulence = 0.5 + 0.5 * sin(6*(xcomp(P)+turbulence)) ;
    /* sharpen peaks */
    turbulence = turbulence * turbulence * turbulence;

    /* map discontinuously to vein colors */
    khi = clamp((turbulence-0.5)*4, 0, 1);

    surfcolor = (1-khi)*Cs + khi*veincolor;
    Ci = surfcolor*(Ka*ambient() + Kd*diffuse(N));
}
```

Example 1: *marble( ) is called whenever a programmer needs the color of the light reflecting from a surface at a given point*

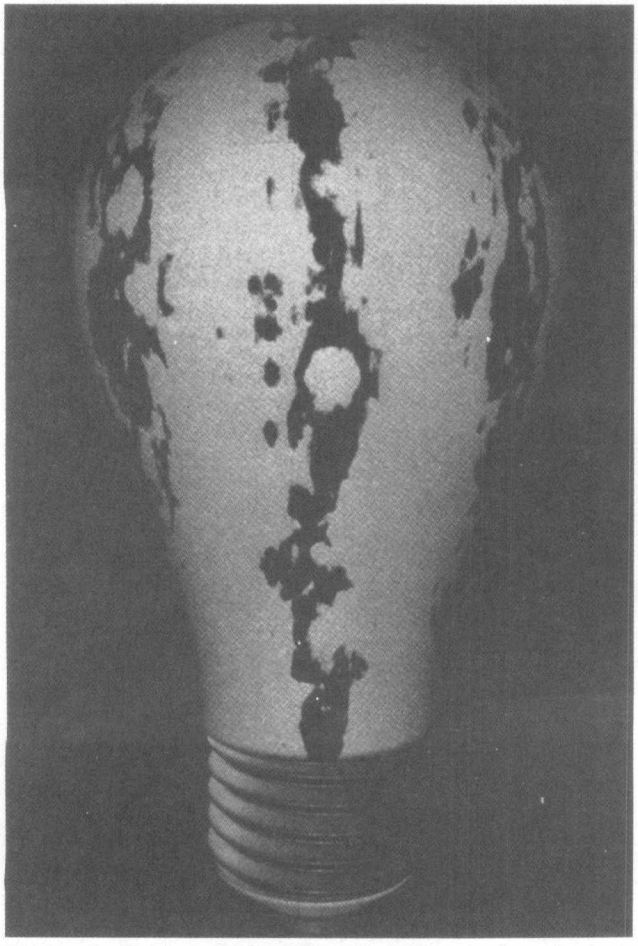


Figure 3: *The light bulb with a "marble" surface shader.*

A shader is invoked in the interface by creating an **instance** of the function, giving values to its instance variables (in Listing One, Kd and Ka). Different instances of a shader (bound, say, to different surfaces) can have different values in their instance variables. For example, the *marble( )* shader might be instanced in a RenderMan program by the routine call

```
RiSurface( "marble", "Kd", .3, RI_NULL );
```

That would give the instance variable Kd the value .3, which it would have every time this instance of the shader was called. Unlike a C function, a shader "initializes" its parameters/ instance variables with a default value, so the shader may be instanced without supplying values for the instance variables. In this example instance, Ka would have .1 as its default value. The RenderMan Shading Language supports the standard C scalar-type **float**, but no others. The only structured data types are built into the language: **color**, giving a light or reflectance color, and **point**, giving a point in three-dimensional space.

A quick glance around any natural environment reveals a mind-boggling variety of textures, reflections, diffusions, and optical special effects

A number of variables used in *marble( )* are neither local to the shader nor instance variables: *P*, *Cs*, *I*, *N*, and *Ci*. They are more important than normal global variables, for they are the implicit parameters of the shader, giving the position in three-space of the point being shaded (*P*), the surface normal at that point (*N*), the reflective color of the surface (*Cs*), and so on. All are set before the shader is called. The output of the shader, the color of the reflected light, is passed back to the renderer in the global variable *Ci*.

The RenderMan Shading Language has predefined extensions in both operators and functions. The functions **faceforward( )** and **normalize( )**, in line 13, for example, are standard in the language. The functions **ambient( )** and **diffuse( )** calculate the contributions of all ambient and diffuse light sources in the scene impinging on the surface point.

Arithmetic operations are overloaded to work on points and colors where appropriate. On line 8 the point *P* is multiplied by 4; the scalar constant is promoted to a point by putting its values in each component of a point before multiplication. Standard control constructs for looping (*while*, *do*, *for*) and conditional execution (*if-then-else*) are included in the shading language, as can be seen on line 17.

A shader has no return value. It communicates its results by setting one or more global variables. The surface shader

(continued from page 22)

calculates the light leaving a surface, and it leaves its result in the global **color** variable *Ci*. If the surface is not opaque, the shader can set the global **color** variable *Oi* to values between *0* and *1*.

Global Variables

The global variables available for access by shaders are listed in Table 1, below.

Surface color and Transparency: *Cs* and *Os* represent the current surface color reflectance and opacity, as defined via the interface from an application program. They may be fixed for each point on the surface (storage class **uniform**) or vary across the surface (**varying**).

Type	Name	Storage Class	Purpose
color	Cs	varying/uniform	Surface color
color	Cs	varying/uniform	Surface opacity
point	P	varying	Surface position
point	dPdu	varying	Change in position with u
point	dPdv	varying	Change in position with v
point	N	varying	Surface shading normal
point	Ng	varying/uniform	Surface geometric normal
float	u,v	varying	Surface parameters
float	du,dv	varying/uniform	Change in u,v across element
float	s,t	varying	Surface texture coordinates
color	L	varying/uniform	Direction from surface to light source
color	Cl	varying/uniform	Light color
color	Ol	varying/uniform	Light opacity
point	I	varying	Direction from surface point
color	Ci	varying	Reflected color of light from surface
color	Oi	varying	Opacity of surface
point	E	uniform	Position of the eye

Table 1: *Global variables available to shaders*

```
light
windowlight(
    point     from =     point (3, 1, -1), /* Center of the window */
              to =       point (0,0,0);
    float     intensity=1,
              order = 2,      /* (# of panes up and down)/2   */
              fuzz = .02,     /* blurred region around panes  */
              framewid = .1,  /* width of a pane frame member */
              panewid = .5;   /* width of the glass in a pane */
    color     lightcolor =     color (1,.9,.6),
              darkcolor =      color (.05,.2,.1);
)
{
    float halfframeplus, halfframeminus;
    float windowwid;
    point wfrom, wto;
    point wL, wP;
    float shade, yabs, zabs, ymod, zmod;

    /* where pane-to-frame transition begins */
    halfframeplus = framewid/2+fuzz;
    halfframeminus = framewid/2-fuzz;
    windowwid = framewid+panewid;

    /* Move window center, sunlight vector back to world space */
    wfrom = transform( "world", from );
    wto = transform( "world", to );
    wL = wto - wfrom;

    wP = transform( "world", P );
    /* Project surface position onto x=xcomp(wfrom) plane */
    wL *= (xcomp(wfrom)-xcomp(wP))/xcomp(wL);
    wP = wP + wL - wfrom;

    /* absolute distance from window center */
    yabs = abs(ycomp(wP));
    zabs = abs(zcomp(wP));
    if( max(yabs, zabs)>(windowwid*order)) { /* Outside window? */
        shade = 0;
    } else {
        /* Modulus reduces insideness to a single pane */
        ymod = mod(yabs,windowwid);
        zmod = mod(zabs,windowwid);
        shade =
            smoothstep(halfframeminus, halfframeplus, ymod)*
            smoothstep(halfframeminus, halfframeplus, windowwid-ymod)*
            smoothstep(halfframeminus, halfframeplus, zmod)*
            smoothstep(halfframeminus, halfframeplus, windowwid-zmod);
    }

    L = to - from; /* always the same (needed for surface shading) */
    Cl = mix(darkcolor, lightcolor, shade);
}
```

Example 2: *A special-purpose light source shader*

Parameter Space: The floating-point values *u* and *v* give the position of the current point on the current surface in parameter space. The **points** *dPdu* and *dPdv* are parametric derivatives, giving the change in surface position *P* per unit *u* and unit *v*, respectively. *dPdu* and *dPdv* are useful for evaluating the rate of change in the surface with *u* and *v*, the surface's parametric velocity. The normal vector *Ng* is just the cross product of *dPdu* and *dPdv*.

Texture Space: The floating-point values *s* and *t* give the texture-space coordinates of the current point on the surface. They may be used to index into a texture map or bump map (see the section on built-in functions,which follows), and so on, for modulating surface properties coherently.

Reflection Variables: A surface shader reports its reflected light back by setting the color variable *Ci*, and reports the opacity of the surface via color *Oi*. The latter is set to 1.0 before the shader is called, indicating a perfectly opaque surface, so the shader needn't bother with it if that condition holds. *marble( )* is opaque, so it only concerns itself with reflected color *Ci*.

Light Variables: The color *Cl* gives the color of light arriving at the surface at a given point, the color *Ol* its opacity, and the point *L* gives the direction from which it is arriving. Although the variables mentioned are always defined when a shader is called, *Cl*, *Ol*, and *L* are redefined for each light source. The **illuminance** construct for surface shaders "loops" over all light sources, setting *Cl*, *Ol*, and *L* once for each light source.

Eye Position: When a shader is called, the point *E* is set to the position of the eye in world space. It is of particular interest for surface shading, because the direction $(P - E)$ gives the viewing direction at the surface point. The *marble( )* function uses only diffuse and ambient reflections, which are independent of viewing direction, so *E* is not used there.

Surface Elements: Two other global variables are provided in support of **surface elements**. Some renderers operate by dividing surfaces up into pieces small enough that it doesn't matter that they are planar, and they can be rendered with a single color. Some shaders may find it handy to know how large a surface element is being shaded. For these purposes the global floating-point values *du* and *dv* give the change in parametric surface parameters *u* and *v* across the element. Of course this assumes that the boundaries of the element form a rectangle in parameter space.

The rest of this article gives an example of three of the most common types of shaders: light source, displacement, and surface.

A Light Source Shader

One of the most endearing aspects of the shading language is that it is simple to write shaders, so you can have fun dreaming up special-purpose shaders. Example 2, this page, shows a special-purpose light source shader similar to one used in the Pixar film *Tin Toy*. That film takes place in a room in a house lit by sunlight through a window; the panes of the window cast bright patches on the scene. The *windowlight( )* shader casts conventional parallel sunlight, masked by the edges and crosspieces of the window. A still from the film is shown in Figure 4, next page. Note how the shadow

(continued from page 24)

from the window falls realistically over the objects in the scene.

The version of *windowlight( )* shown here is instanced by providing a *from* point (the center of the window in three-space) and a *to* point (which gives the direction of the sunlight). The operating assumption is that the window is in a wall in the *x=xcomp(from)* plane in world space. The basic operation is this: a point on a surface being shaded is cast back onto that plane along the direction of the light. The resulting point is subtracted from the window center to get the distance in *y* and *z* between the projected point and the center of the window. If the projected point is within a pane, shade becomes 1 and the output color is light color; if it is not within a pane, shade is 0 and the output is dark color. An interpolation function is used around the edges of the panes to blur the edges of the window.

```
displacement
threads ( float freq = 5.0,
                amplitude = 0.1,
                phase = 0.0,
                offset = 0.0;)
{
    point nDel;
    float scale;

     /* surface displacement occurs along nDel */
    nDel = normalize(N);

     /* The amount of displacement is given by a sinusoid along the
        length of the cylinder, with the place around the cylinder
        providing an additional phase factor for spiral threads */
    scale = (sin( PI*2*(v*freq + u + phase))+offset) * amplitude;

    if( v > 0.95) /* Damp the oscillation to 0 at the ends */
        scale = scale * (1.0-v) / .05;
    else if( v < 0.05 )
        scale = scale * v / .05;

    nDel = nDel * scale;
    P = P + nDel;
    N = calculatenormal(P);
}
```

Example 3: *This shader threads( ) uses a sinusoidal modulation of the surface based on u and v*

Here is a solid use for point-transformation routines. Because shaders operate in a coordinate space which is a priori undefined, and we want to operate in world space (to make the plane of the window predictable), we have to transform points.

Another new item in *windowlight( )* is the appearance of the **smoothstep( )** function. This is built into the shading language; its syntax is

```
smoothstep( min, max, val );
```

All arguments are float expressions. If *val<=min*, **smoothstep( )** returns *0*; if *val>=max*, 1. In between, a smooth cubic interpolation eases between the two. **smoothstep( )** provides the fuzzy shadows in the window light.

The last function in *windowlight( )* is on the last line. **mix( )** performs a linear interpolation between lightcolor and darkcolor based on shade, which should always lie between 0 and 1.

Figure 4 illustrates a number of other effects available using the shading language. The cellophane on the toy box, for example, is a flat surface that was given a wrinkled appearance by a displacement shader. The floor, the design on the box, and the wall all employed texture maps. A special-purpose surface shader was used to provide the window-shaped highlight on the toy; no ray tracing was required.

A Displacement Shader

The light bulb image of Figure 3 used a variety of special-purpose shaders. One of these was a displacement shader, *threads( )*, which displaced the surface of a simple cylinder to form a spiral thread. It used the surface parameters of the cylinder: the global variable *u* gave the relative position of a surface point around the cylinder, and *v* gave the relative

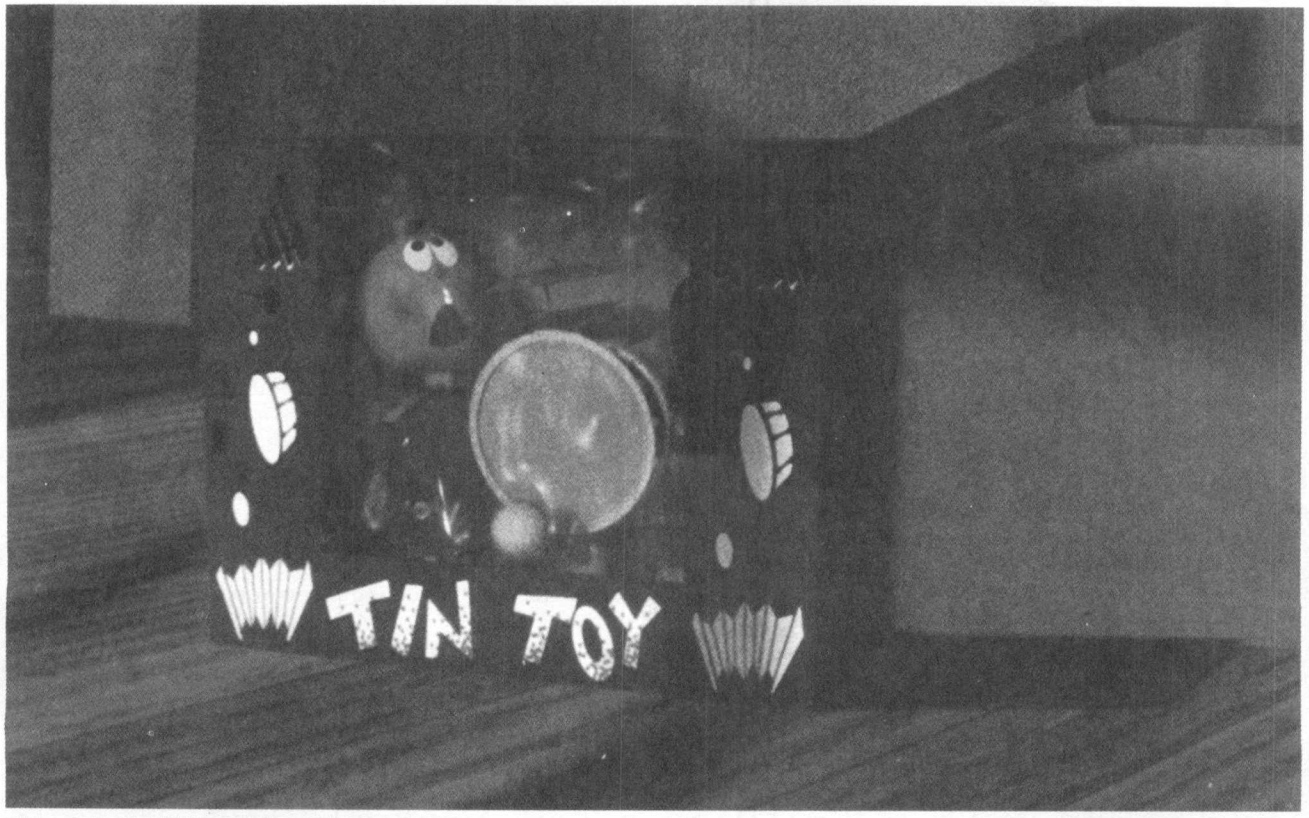


Figure 4: *A still image from "Tin Toy," a computer-generated film*

position across it. Both ranged from 0 to 1. The shader is shown in Example 3, page 26.

The shader *threads( )* uses a sinusoidal modulation of the surface based on *u* and *v*. The primary modulation is in *v*, with the frequency *freq* given as an instance variable. *u* is used as a phase factor, so that the threads really do spiral.

```
surface
filament ( float freq = 5.0, phase = 0.0, width = 0.3;
           color filcolor = color (1, .84, .76); )
{
    float distance; /* Distance from the spiral */

    distance = mod((t*freq + s + phase), 1.0);
    if(distance < width) {
        /* Inside the filament */
        Ci = filcolor;
        Oi = 1.0;       /* Make it opaque */
    } else {
        Oi = 0.0;  /* Make it transparent */
    }
}
```

Example 4: *filament( ) computes the distance from the spiral using global texture coordinates s and t rather than surface parameters u and v*

The RenderMan Specification

The fundamental goal of the RenderMan specification, developed by Pixar and announced earlier this year, is to make it possible for programmers to create photorealistic digital images that are comparable in quality to those produced with a camera. Up to this point most modeling systems have concerned themselves only with the shape (geometry) of an object. The RenderMan specification makes it possible for programmers to more easily address the visual attributes (surface characteristics) of an object as well.

To accomplish this goal the RenderMan three-dimensional scene description interface consists of three parts: 1. The shading language (described by Steve Upstill in the accompanying article); 2. A set of geometric primitives for specifying the shape of objects; and 3. The "glue" for attaching surface attributes (defined by the shading language) to that shape (defined by the geometric primitives). Among the geometric primitives are those for polygons and patches; the shading language provides surfaces, textures, atmospheres, and lights.

The interface procedures are described in C and are accessible from programs written in C. Over the coming months Pixar will release additional interfaces for other languages. Furthermore, the specification is written so that PHIGS, PHIGS+, and other existing graphic standards can be readily interfaced to RenderMan. The specification is device- and display-independent.

So far several major players in the computer-graphic and workstation market have announced support for the RenderMan specification, including Sun, Apollo, NeXT, MIPS Computer, DEC, Symbolics, and Prime, as well as AutoDesk, Industrial Light, and Magic (LucasFilm), Walt Disney, and Pacific Data Images. The last is a leader in television animation and graphics.

The RenderMan specification is available to developers from Pixar for $15. Although the specification document is copyrighted, Pixar will grant free licenses to developers who create products that meet the RenderMan specification. Copies of the specification can be obtained from Pixar at 3240 Kerner Blvd., San Rafael, CA 94901; 415-258-8100. —eds.

There is another explicit phase factor, *phase*, which effectively rotates the threads about the cylinder. The instance variable offset provides a *_DC_* level for the oscillation relative to the cylinder's surface, and amplitude scales the oscillation to control the thread's depth.

For niceness, the oscillation is damped at the top and bottom of the cylinder, converging on 0 at the extremities to provide a perfectly circular opening at either end.

A Surface Shader

Because a surface shader has the ability to manipulate opacity, it has a powerful potential to manipulate the apparent geometry of a surface. For the light bulb picture, I needed a spiral filament, but I didn't want to take the time to define it as some kind of bicubic surface, so I wrote the *filament( )* shader to make a simple cylinder look like a filament.

It uses the surface parameters of the cylinder to create a spiral function on the surface in a way similar to the way the *threads( )* displacement shader works. Rather than modulate the surface, though, it performs a simple test: is the surface point within (instance variable) width of the spiral? If so, the surface is opaque, and given a color which by default approximates tungsten. If not, the surface is transparent.

The *filament( )* shader is depicted in Example 4, page 28. It consists of a cylinder with a cone at either end so that the filament spirals down to a point for joining with its support wires. *filament( )* computes the distance from the spiral using global texture coordinates *s* and *t* rather than surface parameters *u* and *v*. I could have mated the filament on the cones to the cylinder by defining a texture space inside the application to extend over all three objects. Instead I did it by manipulating the phase instance variables until the filaments joined.

The glow around the filament was provided by another special shader. Applied to a sphere (elongated here by a nonuniform scale operation), it simply manipulated the opacity of the sphere to peak in the center and fall off to zero at its edges.

This filament also sags a little in the middle. This catenary "droop" was obtained using a simple displacement shader.

Conclusion

Two kinds of professional programmers can profit from language implementations such as the RenderMan Shading Language. One kind of programmer is the one who needs to reproduce reality for databases of colors and textures for special-purpose applications such as product design, architectural rendering, and industrial material specification. RenderMan standard and shading languages make it reasonably simple to compile, maintain, and use such databases. There are opportunities awaiting both for putting these databases out for the public and for providing material-manipulation front ends for them.

The other kind of programmer who can benefit from the shading language is the one on the receiving end of that process. Given a reasonably easy, uniform way to include complex, interesting light and material characteristics in nominally simple scenes, the realism and visual appeal of computer-graphic images should see a radical improvement.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Putting Graphical Interfaces into PERSPECTIVE

An application without representations is tyranny

by Kent Dahlgren

Inspired by the success of the Apple Macintosh family with its user friendly, windowed interface, the personal computer industry has adopted the use of graphical interfaces as one of its dominant trends. The market is rapidly advancing to a point when advanced graphics will be expected of all new applications. Unfortunately, the proliferation of windowing standards, graphics libraries, and printer interfaces has created confusion among both hardware and software developers. A wide range of interfaces is available with an equally wide range of functionality. In addition, issues of compatibility between various subroutine libraries and windowing interfaces are present. This article provides programmers with an overview of some of the more important graphical interfaces, as well as considerations in selecting one.

Conceptual Model

Consider a model representing the components that might be present in a graphical interface package. This will provide a common frame of reference for the purposes of discussion, the same way that the OSI networking model serves as a means of describing networking interfaces. Figure 1, page 33, shows the components of the model, as well as their relationships.

In addition to the components themselves, consider the terminology describing two of the key interfaces in most graphics system implementations. The Applications Programmer's Interface (or API) is a set of routines that allows an application programmer to communicate with both the Window Manager and the Graphics Engine. Systems programmers who are porting new graphical interfaces to a machine, as well as hardware vendors who are interfacing graphics hardware, are concerned with the Virtual Device Interface (VDI). These represent the front end and back end of a computer graphics system.

Kent Dahlgren is the strategic planner for graphics products at Paradise Systems, a manufacturer of graphics devices. He can be reached at 800 E. Middlefield Rd., Mountain View, CA 94043.

User Interface

In windowing environments, the user interface is commonly referred to as the "look and feel." This is the channel through which the user communicates with the Window Manager. This channel allows the user to alter the size, shape, and arrangement of windows, as well as open and close them. The user interface also handles pull-down and pop-up menus, dialog boxes, and other graphical elements of communication.

Note that the user interface is, for the most part, separate from the remainder of the windowing system. In many systems, the differentiation is purely conceptual. On the other hand, some systems (such as X Windows) don't define any user interface as part of the specification. In those cases, the implementer must decide how screen actions will affect the system state.

As a result of this separation between the User Interface and the Window Manager, you can map a common look and feel onto differing windowing systems. Or, you can map several user interfaces onto a single windowing system. If done properly, such differences between user interfaces are transparent to applications.

The recent popularity of X Windows in the Unix system community has led the development of the Open Look user interface. This interface is being championed by Sun Microsystems and AT&T. The rival OSF Unix camp has been examining contenders for their own X Windows user interface. (There is considerable speculation that OSF is considering the IBM/Microsoft Presentation Manager as its standard user interface.) The ability to tailor a distinctive user interface in order to differentiate between various products in the marketplace is one reason why X Windows is popular with Unix systems vendors.

Window Manager

The Window Manager is responsible for the abstraction of a bit-mapped display image to multiple, virtual display surfaces. It maintains the system's data structures and informs both the user interface and the applications about the size, shape, and visibility of the various windows displayed on the screen. In the case of Microsoft Windows, this is the portion of the system that layers multitasking capabilities on top of DOS.

Two classes of interactions occur between applications and the Window Manager. Applications request the manager's services through a library of subroutines that handle such nondrawing activities as the opening and closing of windows. The Window Manager responds to user-generated events by sending messages to the affected applications. Clicking on a window's close box, for example, sends a close message to the application that owns that window, as well as other applications owning windows uncovered by the closing receive messages that indicate what needs to be redrawn.

This type of asynchronous, event-driven environment requires program structures closer to that of real-time control environments than to typical applications programs. Figure 2, page 33, shows the basic flow of a typical

windowing application as expressed in pseudocode.

Although the structure of the program requires rethinking, the windowing system itself typically offloads many common chores from the application. For example, a single function call within the Macintosh programming environment initiates the opening of a window that allows the user to select a file. Putting this level of functionality in one call not only simplifies programming, but also ensures that the user will encounter the same menu structure in different applications. This uniformity allows Macintosh users to quickly learn new applications.

Graphics Engine

The Display List Manager, Mapping/Translation Layer, and Rendering Interface are collectively referred to as the Graphics Engine. Through its drivers, the Graphics Engine handles the task of putting graphical objects on the screen. Applications use the Graphics Engine to display objects within their windows. The user interface uses it to construct window borders, menus, screen backgrounds, and other visual elements.

In nonwindowed graphical systems, the User Interface and the Window Manager are not present, and the entire interface consists of components of the Graphics Engine. The Graphics Engine itself is also considerably simplified in these cases since there is no requirement to map multiple logical screens to the physical screen.

Display List Manager

The Display List Manager decouples the generation of drawing requests of the application from the hardware (or software) that performs the rendering. This requires some form of buffering, which could be as simple as a queue or as sophisticated as a hierarchical, object-oriented database.

Even a simple queuing arrangement can be beneficial when a graphics coprocessor performs the rendering. A display list queue eliminates the requirement of the host CPU waiting for completion of the first drawing command before issuing the next. Applications tend to issue these drawing commands in bursts; the display list queue evens out the workload.

More sophisticated drawing interfaces (such as GKS, PHIGS, and HOOPS) allow the application to group drawing primitives and manipulate them as single objects. PHIGS and HOOPS carry this idea one step further by arranging these display list groupings into hierarchies that allow children to inherit characteristics from their parents. Inheritance is particularly important in 3-D graphics, where the inheritance allows the programmer to manipulate one component, several related elements, or an entire complex object, all with equal ease.

An example is the image of a robot arm, which might consist of a base, an upper arm, a lower arm, and a hand. All of these parts share a positional relationship to each other. If one rotates the base, all the other components remain fixed with respect to one another and move as a unit. Therefore, the base is the parent node of the hierarchy, and all other parts inherit the attribute of position from the base. Each child component also has some freedom of movement, which affects its children but not its parents. If you move the lower arm, for example, the hand must go with it —obeying the law of inheritance —but the upper arm and base are unaffected.

Mapping/Translation Layer

Most graphical interfaces map drawing primitives to the display through layers of coordinate transformations. Figure 3, page 35, shows how GKS implements coordinate transformations in a two-dimensional space. The coordinates that the application uses to describe objects are referred to as the World Coordinate (WC) system and use a floating-point representation. GKS then maps these points as an internal abstract display using what are called Normalized Device Coordinates (NDC). These coordinates are unsigned values normalized between 0 and 1 in both the X and Y directions. The rendering interface maps NDC to the actual Device Coordinates (DC) of the output medium. This two-stage mapping allows GKS applications to zoom or pan the viewing area over the database by changing transformation parameters.

While the bulk of the older interfaces supports only two-dimensional drawing spaces, the increasing interest in CAD-type software has created a demand for three-dimensional graphics capabilities in new interfaces. The development of personal computers with the power of a workstation has only recently made three-dimensional graphics practical on small systems.

Handling three-dimensional-display lists is not challenging. Rather, the problem is mapping the data to the screen and rendering it in the display buffer. In order to give the WC system sufficient dynamic range, floating-point coordinates normally are used. The transformation of each point requires at least one matrix multiplication (and usually several), and it incurs significant addi-

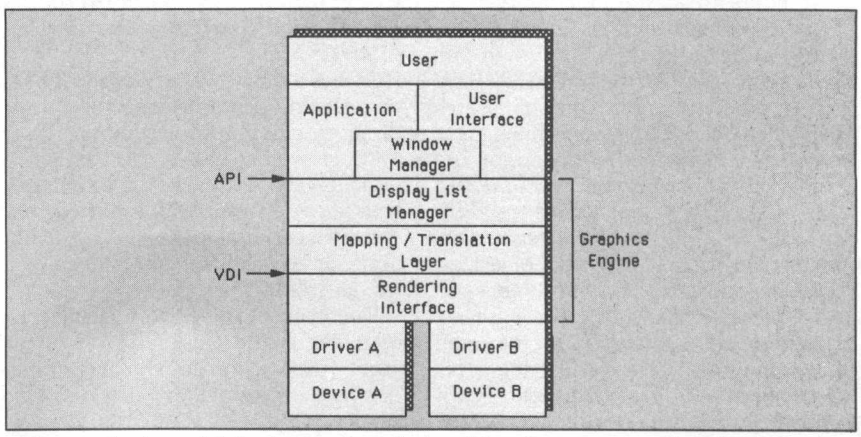


Figure 1: *A model depicting the relationship of components that form a graphical interface*

```
BEGIN
   Initialize data structures
   Setup menus
   Open main window
   WHILE ( not done)
   BEGIN
      CASE ( message type) OF
         type_A_message : execute_A_handler( );
         type_B_message : execute_B_handler( );
         type_C_message : execute_C_handler( );
         type_D_message : done = TRUE
         default handler
   END
   Clean up environment
END
```

Figure 2: *Pseudocode flow of a typical windowing application*

(continued from page 33)

tional overhead for hidden line removal and surface shading. These programmers who use 3-D graphics on the present generation of personal computers must either be content with nonreal-time image creation, or they must invest in expensive special-purpose hardware.

Three-dimensional graphics standards (whether formal or vendor-specified) have several attractive features. For one, the programmer does not have to deal with the complex issues of transforming internal representations of objects into visual images. These tasks are done by making simple API subroutines, passing the names of an object to be manipulated, and various control parameters. For another, the level of data abstraction allows you to achieve performance improvements transparently as new and faster hardware becomes available —assuming, of course, that the new platform supports the graphical system under which the application and its data were developed.

Text Representation

Two methods exists for displaying text in graphics systems. The most common in personal computers is raster text. Fonts are stored in memory as an array of bitmaps indexed by the character value, and then placed in the desired position on the screen by using a copy operation. If the graphics hardware includes logic for performing these memory transfers, the operation occurs quickly. The primary problem with this type of representation is the difficulty in rotating the character images and scaling them to different point sizes.

More sophisticated environments use stroke or vector fonts instead of bitmaps. Here the outline of a character is formed from straight and curved line segments and then filled. The beauty of this approach is that, unlike raster fonts, the characters can be rotated and scaled cleanly and the fonts are not intimately linked to the resolution of the device.

Metafiles

Metafiles provide a means of archiving collections of graphic primitives, either for future use by the same application or for use by other applications (image libraries). Since collections of primitives are inherently device-independent, image libraries can be used by other systems that support the same graphical interface. The key international standard for this capability is the Computer Graphics Metafile (CGM), a specification approved by both ISO (ISO 8632) and ANSI (ANSI/X3.122).

Having covered the general characteristics of the components and issues surrounding graphical interfaces, let's survey the most significant standards in use today.

Graphical Kernel System

GKS was the first formally approved two-dimensional-graphics interface standard (ISO 7942, ANSI/X3.124). It provides a rich set of graphics primitives and a flexible mapping scheme. It also includes facilities for applying geometric transformations to primitives or groups of primitives. Beyond these fundamental concepts, GKS supports object-oriented graphics programming. This is implemented as the recording of a series of calls in a structure referred to as a *segment*, which can then be replayed to reproduce a complex series of operations.

This facility has some serious limitations. Once a segment has been defined, you have no way to edit it. If changes have to be made, you must delete it and rebuild it from scratch. Another limitation is that segments cannot be defined hierarchically, which means that segments cannot contain references to other segments. Later standards (such as PHIGS) remove these limitations.

Programmers Hierarchical Interactive Graphics System (PHIGS)

PHIGS is a draft standard (ANSI/X3.144) developed to fill the need for a three-dimensional graphical interface standard, as well as to correct some of the deficiencies of GKS. Its drawing model was derived from GKS and the syntax of the function calls is similar, but PHIGS goes far beyond the capabilities of its predecessor.

The core of the PHIGS Display List Manager is the Centralized Structure Store (CSS). The CSS is a hierarchical database for maintaining models of graphical objects. The segments used by GKS are referred to as *structures* in PHIGS. Not only has the name changed, but structures can be edited and can refer to other structures. The only limitation on structure references is that recursion is not permitted (that is, an object cannot be defined in terms of itself). PHIGS has the additional capability of using "generalized structure elements," which allow the inclusion of implementation-dependent extensions within the database.

Such tight coupling of the graphical database and the drawing components greatly simplifies the development of applications that must deal with complex objects. For example, you can construct an image of a complete jet aircraft from the specifications of its individual parts. The entire image can be rotated as an entity by using one system call, with PHIGS responsible for translating all of the components. The application is only required to convert the data format of the parts into a PHIGS representation and define their interrelationships.

The downside, of course, is the amount of horsepower required to support such comprehensive functionality. Running PHIGS on anything less than a 68020- or 80386-based platform results in unacceptable performance.

Currently, a set of proposed extensions to PHIGS is collectively referred to as PHIGS+. The chief thrust of these extensions is the addition of shading capabilities. Shading algorithms include both Gouraud and Phong, but not ray tracing. This is consistent with the philosophy of keeping PHIGS an interactive standard, inasmuch as the gigaFlops required to produce ray traced images in real time are not likely to be readily available for several years.

Postscript

Postscript was developed in 1982 by Adobe Systems. 2-D in nature, its imag-

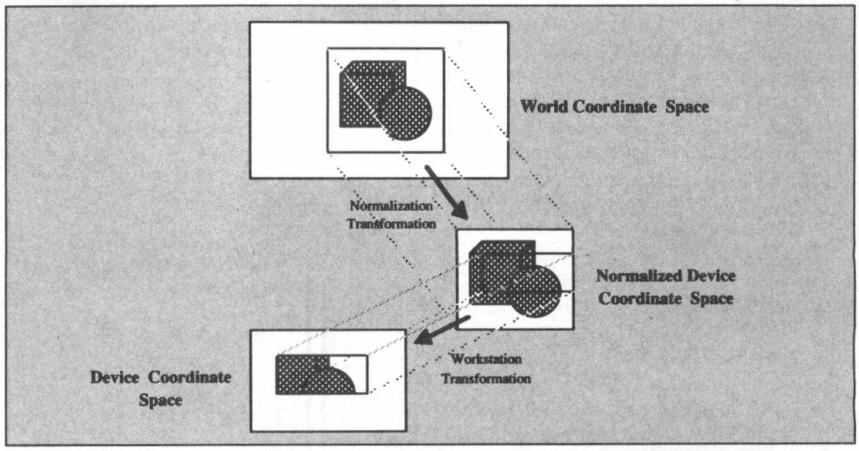


Figure 3: *2-D coordinate transformations in GKS*

GRAPHICAL INTERFACES

(continued from page 35)

ing model is based on concepts derived from the graphic arts. It is entirely output-oriented, and currently has no constructs for user interaction. It is an interpreted language with a Forth-like syntax.

Currently the bulk of Postscript implementations are printer-based. It is a testimony to the language's power and elegance that can be found in everything from the Apple's LaserWriter to Linotronic typesetting equipment. A screen-based derivative is used by Sun Microsystems as the drawing interface for its NEWS windowing environment. The official Adobe version of Display Postscript is also the graphical interface on the recently released NeXT personal computer.

Postscript's approach for handling fonts represents one of the most sophisticated text-rendering interfaces developed to date. Characters consist of Bezier cubic splines, which allow very precise images of complex shapes to be described with a few control points. In addition to this parametric information, Postscript font files contain heuristics rules for translating characters through rotation and scaling. These rules allow the interpreter to correct anomalies that may creep in as a result of rasterization.

The primary factor limiting the growth of Postscript is that, until recently, Adobe Systems was the sole source for ports to new hardware. Several software houses (including Phoenix Technologies) have or will soon be releasing compatible implementations.

Computer Graphics Interface (CGI)

CGI is the ISO/ANSI draft of a VDI specification (ISO DP9636, ANSI/X3.122). It was designed to be the VDI for GKS and its functionality is tailored for that purpose, although considerable efforts were expended to make the interface useful for other higher-level standards. CGI currently supports a rich set of drawing primitives, attribute and segment manipulation functions.

CGI has influenced the development of interfaces from Digital Research, Microsoft, GSS, Nova Graphics, and others. Although these interfaces had their origins in CGI, most have diverged from it. One reason is that few developers are able to wait for a proposed standard to follow the long and winding road to becoming an official standard. Another reason is that many developers want to strip out functionality in order for the implementation to run efficiently on the lowest common denominator (a 4.77 MHz 8088). At this point, the $64,000 question is if these specifications will move closer to CGI as the standard is finalized and higher-performance hardware becomes the norm.

RenderMan

Pixar's RenderMan rendering interface was designed to address the needs of applications that want to present a photorealistic representation of objects. Unlike DGIS and CGI, RenderMan is specifically designed to operate in a 3-D graphics environment. In many areas, RenderMan and the proposed PHIGS+ extensions overlap. Overall, RenderMan is far more sophisticated in its imaging model. This is apparent in its support for ray tracing and numerous shading models. RenderMan's emphasis is on the quality of the image, not on real-time interactivity (as with PHIGS+). In addition, RenderMan makes no attempt to be a complete interactive graphics environment. It does not support user input, text, and non-surface primitives (such as lines and curves).

Macintosh Finder

The Macintosh Finder is the user interface for the Apple Macintosh family of computers. It consists of a set of modules that includes the Window Manager, Resource Manager, Font Manager, Control Manager, Menu Manager, and so on. The Finder handles graphics through a 2-D interface called QuickDraw, which (like the bulk of the system software) resides in ROM. The amount of ROM-based firmware (256K in the Mac II) is one of the reasons no one has cloned the Macintosh. The Finder allows multiple overlapping windows and the recently introduced Multifinder allows limited multitasking. Graphical information may be transferred between windows through the use of the clipboard, which can be thought of as a graphical paste buffer.

With the Mac II, Apple introduced an enhanced version of the Rendering Interface called Color QuickDraw. As the name suggests, the major improvement is enhanced color support. The original QuickDraw interface supported only eight colors, which was seldom a limitation since earlier Macs had a monochrome display. With Color QuickDraw, applications specify colors from a 48-bit color space. The system firmware is responsible for mapping these logical colors to physical colors on the screen.

Another feature of the Mac II drawing environment is seamless support for multiple-display adaptors. All of the screens attached to the system are logically concatenated to form a single dis-

play space. The user can drag windows between displays and can even have windows straddle screens. The logical color space makes differences in the color depth of the physical displays transparent to the application. From the application standpoint, the elegance and simplicity of this approach is only matched by its complexity from the implementation viewpoint.

Windows/Presentation Manager

After the Macintosh Finder, the most prevalent windowing interface for personal computers is Microsoft Windows. Unlike the Finder, Windows has had nonpreemptive multitasking capabilities built into it from the beginning. Although its user interface intentionally differs from Finder in numerous ways, an experienced Macintosh user can quickly become comfortable in Windows. In fact several applications (including Microsoft Excel and Aldus Pagemaker) are nearly identical under the two interfaces.

To complement its multitasking capabilities, Windows supports an interprocess communications protocol called the Dynamic Data Exchange. This protocol is like X Windows in that it is based on a client-server model. It differs from the X protocol in that it is used for both graphical data and general-purpose interprocess communication. The reason for such a difference is that Windows runs under DOS, a single-tasking operating system. On the other hand, X Windows was designed to be hosted by the Unix system, which has a complete set of interprocess communications utilities already in place.

The Windows Graphics Interface is known as the Graphics Device Interface (GDI). It is a 2-D interface that uses a one-stage logical-to-device coordinate conversion. Like QuickDraw, Windows has a somewhat limited set of drawing primitives, but it does feature very powerful and flexible raster operations.

The most noteworthy aspect of Windows is its driver interface. The designers of Windows were faced with supporting devices as simple as the CGA and as complex as the PGA with its graphics coprocessor. The challenge was to design a driver interface that would allow the vendors of simple display controllers to write simple drivers with a few capabilities (since every function has to be performed in software), while at the same time be able to take advantage of highly sophisticated graphics devices.

The solution was to create an interface that requires any driver to perform only a small number of essential functions. Beyond this core is a wide range of functions that the driver can optionally support. When the GDI wants to perform an operation, it checks a data structure that indicates the services the driver supports. If the particular operation is supported, the GDI calls the driver directly, and otherwise it emulates the function through calls to other services that the driver does furnish. For example, if the driver doesn't support circles, the GDI constructs a circle via multiple calls to the line-drawing functions (which is required).

The X Window System

X Windows originated as part of Project Athena at MIT. It was developed with the help of several computer manufacturers and has become the windowing interface of choice for *Unix workstations*. It is supported (in one form or another) on systems produced by DEC, HP, Apollo, Sun, and others. Because the drawing interface for X is fairly minimal, most implementers have extended it, usually by the addition of a supplementary interface.

One of the most significant contributions of X is client-server architecture, which enables nodes to transparently interchange graphical information over a network. This is one of the architectural features that differentiates the X Window system from Microsoft Windows and the Macintosh environments. When an application opens a window under X, it specifies the network address. The system automatically routes the graphics output calls made by the application running on one node to the node that will do the actual rendering. Of course, a high-performance network is required to handle all the traffic resulting from such a facility. For this reason, such a feature will not likely be added to any of the PC-based windowing environments in the immediate future.

Selecting a Graphical Interface

The choice of a graphics system depends on a number of factors that attempt to match as closely as possible capabilities with known and probable requirements. For example, if the primary purpose is to provide a consistent user interface through the use of windows and menus, Microsoft Windows or Finder are more appropriate than, say, Postscript or HALO, which have no explicit user interface support. On the other hand, if highly realistic three-dimensional graphics is the chief goal, something like PHIGS or RenderMan is a better choice. Table 1, this page, lists the major features of several commercial graphics interface systems.

Support for various hardware/software platforms and data portability among them might also be an important consideration in selecting a graphics system. For example, a work group using both '386-based PCs running DOS and Sun workstations under Unix and must share data and programs. In that case, Finder is definitely out of the running, since it's only available on Macintosh machines, but something like the widely-implemented HOOPS might be ideal.

Another very important factor is ease of programming and the quality of documentation. One of the chief objectives in using a packaged graphics interface is productivity: relieving programmers of the complex tedium required to get images onto the screen or manage the user interface (or both). A well-designed API allows the programmer to concentrate on the purpose of the application, rather than on display management, thus achieving the goal of increased productivity. One that is poorly designed or —even worse, badly documented —merely replaces one set of complexities with another.

Graphical interfaces are powerful and complicated toolsets. The key to selecting a graphics package and using it effectively is knowing what components it has and how they interact to solve your programming problems.

DDJ

Interface	Originator/Implementor	Windowing	3D	Text	Structures	Metafiles	Raster Ops
GKS	Graphics Software Systems	N	N	Stroke	Y	Y	N
PHIGS	Template	N	Y	Stroke	Y	Y	N
PostScript	Adobe Systems	N	N	Stroke	Y	Y	Y
HALO '88	media cybernetics	N	N	Stroke/Raster	N	Y	Y
DGIS	Graphics Software Systems	Optional	N	Stroke/Raster	N	N	Y
PGL	Vermont Microsystems	N	Y	Raster	N	N	Y
8514A/AI	IBM	N	N	Stroke/Raster	N	N	Y
MetaWINDOW	Metagraphics Software Corp.	Y	N	Stroke/Raster	N	N	Y
Quickdraw	Apple Computer	Y	N	Raster	Y	Y	Y
GDI	Microsoft	Y	N	Raster	N	Y	Y

Table 1: *Major features of several graphics rendering packages*

IMAGE COMPRESSION VIA COMPILATION

A graphics coprocessor is one way graphics intensive jobs can be done in less time

by Victor J. Duvanenko

Image compilation is a unique method of image compression whereby an image is transformed into a graphics coprocessor instruction stream. Even a simple implementation of the image compilation technique has lead to a 2.5:1 compression ratio with a corresponding reduction in the image reconstruction time.

At present, resolutions for graphics displays vary from 320 × 200 on the very low-end to 2K × 2K on the high-end, with 640 × 480 being fairly common. A 640 × 480 display consists of 307,200 pixels. If the depth is 8 bits per pixel (a byte for each dot on the screen), then it takes 307,200 bytes to define a full screen image. This is a considerable amount of data to move or manipulate interactively. Considering that resolution of raster displays is increasing every year, the job of graphic data manipulation will not get any easier.

To put into perspective what it means to manipulate large amounts of data, consider that to transmit 307,200 bytes of data would take 43 minutes over a 1200-baud modem and about five minutes over a 9600-baud modem.

At 9600 baud, you could not manipulate an image interactively. While you could manipulate text, it would be impractical to manipulate high-resolution bitmaps. With high-resolution color scanners rapidly becoming an inexpensive reality, interactive image manipulation will be a necessity.

With applications such as large graphics databases, any compression is welcome. With other applications (such as interactive graphics systems) bitmap reconstruction time is of paramount importance. Consequently, any reduction in bitmap size would directly translate into improvement of response time.

You can reduce transmission time to enable more efficient graphics data manipulation by using the following methods:

1. Use a faster link (Ethernet perhaps) to the graphics terminal. At 1 Mbyte per second, it would take about 300 milliseconds to transmit a bitmap. Since Ethernet boards cost anywhere from $300 to $1,000, this is an expensive alternative.
2. Use a graphics workstation with a local hard disk. Consider that it takes 3.5 seconds to load a 640 × 480 bitmap from a hard disk. (This was timed on a 6 MHz Xenix 311, using low-level I/O read function of C, the 80286 used a movsw instruction, disk I/O being the limiting factor.)
3. Use a RAM disk or a graphics memory for local bitmap storage instead of hard disk. This would enable you to fit about three full bitmaps per megabyte of storage. (Keep in mind that to store two minutes of footage takes about 960 Mbytes.)
4. Make your graphics terminal smarter so that a higher-level picture description is transmitted or stored on the hard disk, laser disk, or in graphics memory. This requires a dedicated high-performance graphics engine to reside locally to interpret the high-level commands and to reconstruct an image quickly. Today's microprocessors are much too slow for such tasks and do not have instruction sets that are well-suited for graphics.

One such graphics engine is the Intel 82786, a chip that contains many graphics instructions/primitives embedded in hardware. The 82786 also provides a 40-Mbyte/second bandwidth to graphics memory (a power that will be discussed later).

The programs described in this article are designed to illustrate the specific advantages provided by a dedicated graphics coprocessor like the 82786. The image compilation code consists of two stand-alone programs — one program for image compression and the other for image reconstruction.

The compression is performed entirely by the microprocessor, which converts (compiles) an image/bitmap into a series of higher-level graphics instructions. The graphics program, not the bitmap, is stored on the hard disk. To reconstruct an image, the CPU loads the graphics program into graphics memory. The graphics coprocessor then executes the code and the bitmap is perfectly restored.

Parallelism can be harvested for improved performance. The graphics program can be split into subprograms. As the CPU loads one subprogram, the graphics coprocessor can be executing another, thus reconstructing an image piece-by-piece in parallel.

If this method of data compression is so great, you might ask, why isn't it being used more widely? Several reasons can be given for this. For one thing, inexpensive dedicated graphics hardware is relatively new. People are just beginning to appreciate the power and flexibility that these machines offer. The cost of these machines may still be prohibitive for some applications. Another reason is that compression takes time —the bigger and more complex the bitmap, the longer it takes. In some cases, this outweighs the gain. Yet another reason is that it takes time, money, and programming skills to refine and solidify compression algorithms. (It took two weeks to write the software in Listings One and Two, pages 82 and 88.) Finally, data compression does not work in all cases. Any general-

Victor Duvanenko is an IC design engineer. He has worked on several graphics chips at Intel. Presently he is a consultant at Silicon Engineering Inc. in Santa Cruz. He can be reached at 1040 S. Winchester Blvd., #11, San Jose, CA 95128.

purpose compression method must make some files longer (otherwise you could continually apply the method to produce an arbitrarily small file).[1]

However, you could always apply the compression method, then see if the file gets smaller. If it does not get smaller, you could then stay with the original. Therefore, the resulting file is always less than or equal to, in size, to the original file.

The Compression Program

The compression program shown in Listing One, expects an 8-bit-per-pixel 640 × 480 bitmap to reside in graphics memory (graphics memory is mapped into part of system memory space). The only necessity is graphics memory accessibility to the CPU.

The easiest (and most obvious) way to compile an image is to:

1. Start with pixel 0 in scan line 0 of the bitmap
2. Look for color 0
3. Encode color 0 runs as the bitmap is scanned line by line.

This procedure would be repeated for all possible colors (in this case 0 through 255). The bitmap will be scanned as many times as there are possible colors. This could get out of hand with 2K × 2K 32-bit-per-pixel bitmaps. Even with 8 bits per pixel, scanning a 640 × 480 bitmap 256 times takes over an hour. The obvious way is usually not the best way.

The next most obvious approach is to process only the colors that the bitmap contains. This requires an overhead of a single pass through the bitmap (one would have an array of flags that would be set, once a color was detected). This helps, but more can be done.

Because the x and y coordinates are always known during the image scan, it is easy to find the maximum and minimum x and y coordinates for each color. A smaller region could be scanned during the compilation stage. This is exactly what the *find_all_colors* procedure of the compression program does. It scans through the bitmap once and establishes the region of color existence. A special structure, *color_struct*, handles this. Each element in the structure contains the maximum and minimum x and y coordinates and the number of times that the color is detected during the scan. To make life easier, a type *color_t* of *color_struct* is defined.

The next stage is to compile each of the regions of color, the *extract_scan_lines* procedure. Adding some initialization instructions is appropriate since you are making a graphics coprocessor instruction stream. Place a define color and a scan lines instruction (see sidebar for a discussion of the scan lines instruction) into a buffer. The array of scan lines will follow. Then proceed one scan line at a time, starting at minimum y and x coordinate for this color.

The run-length encoding is quite simple: as the desired color is found, a flag is set. If the next pixel is of the same color, the count is incremented. If it is not of the same color, the end of the run has been reached and its length is stored in the buffer. This goes on, line-by-line of an image, until maximum y and x have been reached. This procedure is repeated for each color in the image.

The graphics coprocessor instruction stream is stored in an array called *buff*. This array is several disk blocks in size. When it is filled, an end instruction is added and the array is written to a file on hard disk.

The Loader Program

The loader program, found in Listing Two, page 88, expects a file to be in a specific format, which "compact" conforms to. This format consists of packets of data that are preceded by a header. The header describes the address in graphics memory where the data is to be placed and how many bytes of data are to be placed.

The loader program performs some initialization tasks before loading the graphics coprocessor instructions into graphics memory. It waits for the graphics coprocessor to finish any instructions that may still be executing. It then loads the first packet of graphics instructions (scan lines) and directs the graphics coprocessor to execute them. While the graphics coprocessor is executing these instructions, the main processor gets another packet from the disk. This process continues until all packets have been loaded. The loader takes a time stamp before and after the loading process and reports the total time that it took to load and reconstruct the bitmap.

Benchmark Results

To further illustrate some of the performance advantages that a graphics coprocessor provides, Table 1, page 43, lists the test results. These tests were run on Intel's Xenix 311 system (which uses a 6-MHz 80286 microprocessor) with an 82786 add-in board (20 MHz with 1 Mbyte of graphics memory). All bitmaps were 640 × 480 at 8 bits per pixel. Bitmaps 1 – 3 are of Mandelbrot fractals. Bitmap 1, which consists of a straight uncompressed bitmap, is the control. Bitmap 4, a solid, single color bitmap, illustrates the best case.

After reviewing the test results for bitmaps shown in Table 1, you may think that 10 minutes is an excessive price to pay for a 2 1/2 times reduction in size and reconstruction time. It is for some applications. For other applications, it allows them to place 2 1/2 times more information on the storage medium to present 2 1/2 times the amount of data to the user in a given amount of time or to retrieve images 2 1/2 times faster. This could make or break an application. Effectively, an 8-bits-per-pixel bitmap has been reduced down to about 3-bits per pixel, without losing any information.

Some of the routines developed are useful for digital image processing. For example, *find_all_colors* routine generates a profile of the bitmap by quoting the number of times a color appeared in the bitmap. This could be turned into the probability density quote by dividing that count by the total number of pixels in the bitmap. This could be taken one step farther. Based on the probability density quote of a color, you could decide that the color is rare in a bitmap and is not worth processing. This could reduce the compression time as well as compressing the file, at a cost of losing some information. Using this method, you could minimize the picture content that is lost.

In any case, the compaction algorithm speed could be improved (about three times by my estimates). If you used a 20-MHz 80386, the compression time would dip to less than a minute.

Conclusion

This image compilation technique offers infinite avenues for further exploration. This article has explored only a single instruction. Other instructions (*bit_blit*, for example) offer other possibilities. The graphics coprocessors that are presently available have a great

Continued on page 47

	Compression Method	Compression Time	Reconstruction Time	File Size	Compression Ratio
Bitmap 1	none	n/a	3.5 sec.	307K	1:1
Bitmap 2	scan lines	612 sec.	1.3 sec.	120K	2.5:1
Bitmap 3	scan lines	614 sec.	1.3 sec.	121K	2.5:1
Bitmap 4	scan lines	11 sec.	0.02 sec.	2.9K	105:1

Table 1: *Test results of image compilation routines*

The Scan Lines Instruction and Run-Length Encoding

One of the most powerful 82786 instructions is *scan lines*, which is used to draw a series of horizontal lines. It is the fastest 82786 drawing instruction, executing at up to 2.5 million pixels per second, at 8-bits per pixel (faster at lower pixel depths). This instruction is generally used for area fills, and has pattern capabilities.

The *scan lines* instruction looks like this: 1. the instruction itself, 16 bits (0BA00H); 2. the address of the *scan lines* array, a 32-bit value, low word followed by the high word; and 3. the number of horizontal lines to be drawn.

The *scan lines* array elements have a particular format, consisting of: 1. dx, an offset from the beginning of the previous line in the x direction (16 bits, negative or positive); 2. dy, an offset from the beginning of the previous line in the y direction (16 bits, negative or positive); and 3. the length of the line (16 bits, negative or positive).

The *scan lines* instruction starts at the present graphics location, adds dx and dy values of the first array element to it, and draws a line of the specified length. The graphics location is left at the beginning of the line. The new graphics location is then adjusted by dx and dy values of the second array element. The second line is then drawn. This process is repeated until the specified number of lines have been drawn. You could thus draw a scan line at the bottom of the screen and then one at the top of the screen, all within the same *scan lines* array.

Areas of any shape can be drawn by using the *scan lines* instruction, and they do not have to be contiguous. You can jump to any position on the screen and continue drawing at any time, all with a single array (as long as the color and the pattern can remain constant).

You can break up an image into areas of constant color and use a single *scan lines* array to describe each of them (one for black, one for white, one for a shade of green, and so forth). It would take as many *scan lines* arrays as there are colors in an image to completely describe that image.

The *scan lines* instruction closely resembles run-length encoding. Run-length encoding removes redundancy from a data file by replacing data elements with the count followed by the element itself. For example, a sequence of AAAAAA can be replaced with 6A, resulting in 3:1 compression. The same technique can be applied to images. A sequence of six black pixels can be replaced by the number 60 (where 6 is the count and 0 is the color black). For an area of constant color, specifying the color along with every run is redundant. You can specify the offset from the current graphical location followed by the run-length.

This should begin to resemble the *scan lines* definition —dx, dy, and length. For many images, encoding areas in this fashion is an efficient way of removing redundancy, thereby compressing an image. The less space an image takes, the faster it can be transmitted or retrieved from storage.
—V.J.D.

GRAPHIC COMPRESSION

(continued from page 43)

variety of instructions, and this method of image compilation can lead to an adaptive image compression.

It is possible to predict whether an image can be compressed by using run-length encoding. An average length of a run can be calculated for an image. If the average run takes up more space than one scan lines array element (dx, dy, and length use 6 bytes) then run-length encoding is worthwhile. For example, at 8 bits (one byte) per pixel, the average run for an image must be greater than 6 pixels. This is the break-even point for a compression technique. The compression ratio can be improved by removing redundancy from the compressed file. This can be accomplished by encoding repetitive scan lines sequences in the 82786 macros/subroutines. This achieves a two-dimensional compression.

Another method is to reduce the number of bits used for dx, dy, and length elements in the scan lines array by scanning through the array and determining the maximum values ever used. This would require extra processing by the CPU, which would increase encoding and reconstruction time. A differential encoding technique could also be used to encode the differences between successive dx, dy, and length values.[2] This also requires some extra CPU processing. By using combinations of these techniques, you could increase compression ratios to above 10:1. The penalty of added CPU processing time may still outweigh the benefit of reduced I/O time caused by improved compression.

Compression techniques are still in the infancy, and many promising techniques are emerging, such as "Chaotic Compression" (see *Computer Graphics World*, November 1987) promises a 10,000:1 compression ratio.

References

1. Sedgewick, Robert. *Algorithms.* (Menlo Park: Addison-Wesley, 1983), pp 283-294.
2. *82786 Graphics Coprocessor User's Manual.* Intel, 1987.
3. Gonzalez, Rafael C. and Wintz, Paul. *Digital Image Processing.* (Menlo Park: Addison-Wesley, 1987).

DDJ

(Listings start on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

Dynamic Run-Time Structures

A new way of taking care of old business

by Todd King

In some instances, an application must deal with data that can have almost any structure. This situation is especially true with database applications where a set of access routines perform the task of extracting data from a database, and making that information available to the application. This involves two interfaces: one between the access routine and the database, and another between the application and the access routine. In most cases, the access routines must be general enough to handle a wide variety of data structures, yet robust enough that the data can be reliably acquired. The best way to meet these criteria is by designing a simple method for data access that serves as a natural extension of the programming language. This article looks at the way to extend the structure concept to do just that.

In general, a structure is a contiguous block of memory that is divided into elements. Each element has a specific data type, possibly a dimension, and a name. When a program that uses structures is compiled, the structure and any reference to its elements are compiled into pointers and offsets. All information about the name assigned to each element is lost. After a particular structure is compiled, its layout becomes fixed and cannot be changed at run-time. So by using conventional structures, a particular application could only support one data format. While this limitation may be fine with some applications, we generally want to have a single application that can deal with any structure. This application is possible if an application can define a structure and the names of its elements at run-time: that is, if there can be dynamically defined data structures with dynamically assigned element names.

Todd King is a programmer/analyst with the Institute of Geophysics and Planetary Physics at UCLA. He is also associated with the NASA/JPL Planetary Data System project. Todd can be reached at 1104 N. Orchard, Burbank, CA 91506.

Requirements of Dynamically Defined Data Structures

To define a structure dynamically at run-time, an application must perform a few tasks. First, there needs to be a method of specifying elements of the structure, what type each element is, and the name it is to be referenced by. Second, there must be an area of memory set aside for placing the contents of the structure and a method to load the data into the area. Finally, the application must have a way to extract the value associated with a particular element.

To illustrate how to implement dynamic structures, this article examines a special case of structures. In this article, the data types will be limited to those of integer (*int*), *float*, and *double*: all elements will be limited to a single value (no arrays), the element names will be strings of 12 characters or less, and no structure can exceed 1,024 bytes in length. These limitations are self-imposed so that the example in this article will be as clear as possible. Also, the example is designed to be used with databases since database applications typically require dynamic structures to be as highly adaptive as possible. This article describes each step of structure definition and use.

The Data Format

The functions presented here expect the structure definition and the data that the structure is to contain to be in separate external files. The structure definition is in a file with an extension of *.def*, the data is in a file with the extension of *.dat*. The two files are associated with each other by a common base name. For example, the file testdata.def would contain the structure definition for the data in the file

testdata.dat. This type of physical separation is required for relational databases because each type of information is distinct and different.

The structure definition file has the format

```
<type> <element name>
```

where *<type>* is limited to one of *int, float, double.* Specifying any other type simply results in an element value of "unknown type." The *<element name>* can be any name you chose. The length of the name is limited to 12 characters. Only one element definition is allowed per line. The entire file defines only one structure. Think of the base name of the file as the name of the structure.

The data file has a format such that each line of the data file contains all the data for a single structure. The value for each element is specified in terms of textual numbers, with each number separated by a space. Since the definition and data files are textual, it is easy to edit, modify, and enter values.

The Functions

You can define. load, and access a dynamic structure with just four functions. This article discusses functions separately. Listing One, page 92, contains the dynamic structure header file that all the functions use. Listing Two, page 92, contains the routines.

Loading the Structure Definition

To use dynamic structures, the first step is to load the structure's definition. You can load this definition with the function

```
load_v_def(v_file, d_struct)
char v_file[ ];
D_STRUCT *d_struct
```

which interprets the structure definition and prepares the structure for data loading. This function expects the base name (*v_file*) of the files that contain the structure definition and the data. The structure-definition file must have the extension *.def* and the data *.dat.* The file first reads the definition file. The definition declarations look like a C structure definition with the structure wrapper removed with only one variable allowed per line. Table 1, this page, contains a sample definition file. As *load_v_def( )* reads each line, it extracts the type and name portions. The type is then encoded with a call to *v_type( )*, which compares a textual type to all the allowed types. The type and name are stored in the structure-definition variable, *d_struct*, which is passed to the function.

After scanning through the definition file, *load_v_def( )* opens the data file. The file pointer is then stored in the definition structure for use by subsequent calls. This function returns the number of elements defined in the structure or –1 if any failure occurs. A sample call is

```
printf("%d elements \ n",
        load_v_defs("example1"));
```

which loads in the definition of the structure contained in the file *example1.def* and opens the file *example1.dat* for the structure data may be read.

Database applications typically require dynamic structures in order to be as highly adaptive as possible

Reading a Structure

After a structure is defined and the data file opened, the structure data may be read with this call:

```
read_data(d_struct)
D_STRUCT *d_struct
```

This call reads a single line from the data file associated with the dynamic structure *d_struct* and stores the line in a reserved data space. Once a dynamic structure is loaded with data, the value for any element can be retrieved. Because these routines are designed to work with databases, *read_data( )* is usually called multiple times. Each time it is called, the structure is loaded with the next structure of data. The function returns *NULL* when there is no more data available; otherwise, the function returns a pointer to the data space for the structure. Table 2, this page, lists data that works with these routines.

Accessing Structure Elements

The value associated with an element in a dynamic structure can be accessed with this function:

```
get_v_value(v_name, d_struct)
char v_name[ ];
D_STRUCT *d_struct;
```

Each element in a dynamic structure has an associated name. Since the name is represented as a string of characters, a string with the element name in it is passed to *get_v_name( )* each time the value for an element is needed. The function then searches all the elements of the structure *d_struct* for the name. If the function finds an element, then the data associated with that element is extracted from the data space. Since we chose to limit the example to numeric types, the data is transformed to a double variable. This transformation is made so that only one function is needed to access all numeric types.

Completing the Read

After completing a scan through the data file, the data file associated with the dynamic structure should be closed. The function

```
int count
float x
double y
float z
```

Table 1: *Example structure definition*

1	10.0	20.0	30.0
2	32.0	64.0	128.0
3	9.0	18.0	36.0
4	0.1	1.0	10.0
5	2.0	4.0	8.0
6	10.0	100.0	1000.0
7	12.6	14567.89	123.89
8	56.7	12598.90	234.75
9	27.9	1234.90	123.6

Table 2: *Data which can be used with the data structure definition in Table 1*

```
end_read(d_struct)
D_STRUCT *d_struct;
```

performs the close of the file pointer associated with the dynamic structure *d_struct.*

Using the Element Values

An example of how to use the values of the dynamic structure is the function

```
print_v_value(v_name, d_struct)
char v_name[ ];
D_STRUCT *d_struct;
```

Access routines must be general enough to handle a wide variety of data structures, yet robust enough that the data can be reliably acquired. The best way to meet these criteria is by designing a simple method for data access that serves as a natural extension of the programming language.

This function prints the value associated with the element with the name *v_name*, which is the dynamic structure *d_struct.* The function looks at the type of the element, formats the value accordingly and prints the value into a field that is (at least) 16 characters wide. This function checks if the element is declared and if the declared type is supported. If a valid element is requested, a call to *get_v_value( )* is made. The return value from *get_v_value( )* is typecast to its declared type to remove any ambiguity for *printf( ).* If the return value is assigned to a variable, then the value would automatically be typecast by the action of the assignment. Since *get_v_value( )* returns a double, the value may be assigned to any numeric type without any loss of precision, regardless of the elements numeric type.

DYNAMIC RUN-TIME

The Sample Program

The sample program in Listing Three, page 93, uses all the functions presented here. Its purpose is to print to the screen, in a simple tabular format, the requested elements of a dynamic structure. The structure to use and the element names to print are given on the command line. The syntax following the program name is

```
<d_struct> <element_name>
                    [<element_name> ...]
```

where <*d_struct*> will be the base name of the structure file and <*element_name*> can be any name you choose, whether it is in the structure or not. If the given <*element_name*> is in the structure, then the value associated with it is printed, if <*element_name*> is not the structure, the phrase "Unknown element" is printed. If <*element_name*> is in the structure-definition file and the element is specified as a type that is not supported, the phrase "Unknown type" is printed. The example scans through the data file and displays as many lines as there are structures in the data file.

Extensions of The Concept

With a little work, you can make the routines in the example even more useful. One immediate addition is to rewrite the routines to handle binary data. Even with binary data, the structure-definition file should be text, which allows for readable self-documented data. Also, adherent to relational models, the structure definition, and data should be separate files. Other additions include changing the names of elements, allowing the element values to be written (in the example, they're read only), and adding new data types.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Mapping DOS Memory Allocation

Knowing how DOS manages your PC's memory saves you time and trouble

by Robert J. Moore

The concept of a memory control block (MCB) was introduced in MS-DOS, Version 2.0, as the operating system's basic method of tracking memory allocation for application programs and installable device drivers. In this article, I'll discuss how DOS uses memory control blocks and will present a Turbo C program named MCB.C that prints out DOS' current state of memory allocation. First, it might be useful to review how the PC uses memory.

Figure 1, next page, shows the major aspects of the PC memory map as documented by Microsoft and IBM. Addresses are shown in hexadecimal *segment:offset* form. Numeric addresses give absolute memory locations for those items that are always invariant in a PC running under MS-DOS, Versions 2.0, and later. Symbolic addresses vary according to the version of DOS you are running and the number and types of applications presently in memory. More than one program will be in memory if one or more terminate-and-stay-resident (TSR) program has been installed.

Robert J. Moore is a senior project engineer with Hughes Aircraft Radar Systems' F-14 Program in El Segundo, California. He can be reached at 2126 Prosser Ave., Los Angeles, CA 90025.

The program MCB.C in Listing One, page 94, computes and prints out specific addresses shown as *mmmm:mmmm* in Figure 1, which includes most of the unknowns in the DOS memory map. The only address it cannot compute is the starting address of the transient portion of COMMAND.COM (denoted by address *xxxx:xxxx*). MCB.C also provides insight as to how the DOS memory management functions do their job and can show you exactly where all your memory has gone the next time you run out of it on a machine overloaded with TSRs. Finally, MCB.C should give you ideas about how to reallocate memory appropriately should it become necessary. For now, I'll discuss DOS memory control blocks and describe where they fit it into all this.

Memory Control Blocks

Microsoft and IBM document the fact that they use MCBs, but neither documents specific details of how they use them. The contents of an MCB, also known as an arena header, are illustrated in Figure 2, page 57. An MCB contains three fields called the chain id, process ID (PID), and block size. Together these fields occupy the first 5 bytes of the MCB; the remaining 11 bytes are unused. An MCB always begins on a paragraph boundary and occupies exactly one paragraph (16 bytes) of memory. The chain id byte indicates whether this is the last MCB (value *Z*) or if another follows (value *M*). The memory block controlled by a given MCB always follows immediately, starting at the next paragraph, and has a size equal to the number of paragraphs specified in the block-size field of the MCB. I'll refer to these blocks of memory as memory blocks (MBs) to distinguish them from the MCBs. Subsequent MCB/MB pairs always start just after the previous pair. MCBs can be thought of as a forward-linked-list data structure, with the MBs they control neatly sandwiched in between. In normal operation, there are no gaps in this structure. If DOS detects any corruption in the MCB chain, it prints out an error message and halts the system. (I have had occasion to break the MCB chain deliberately during my investigations; DOS doesn't always notice.)

The first MCB/MB pair allocated by DOS is always owned by IBMDOS.COM/MSDOS.SYS and is the fifth PC memory region, as shown in Figure 1. The number of remaining MCBs depends upon whether or not any TSRs have been installed. Typically, any program (transient or TSR) in memory owns at least two MCBs —one for its copy of the

environment and the other for its code and data. If the program has allocated additional space (for example as internal buffer space), it may own more MCBs. If a TSR has freed its environment block (as some do), it may own only one MCB.

DOS Memory-Management Functions

Starting with Version 2.0, all versions of DOS contain functions to allocate memory (function *48H*), free allocated memory (function *49H*), and modify allocated memory (function *4AH*). None of these functions deal directly with MCBs; instead they deal with the associated MBs. Changes caused by use of these functions, however, are reflected in the MCBs, as can be seen upon inspection (using my program) after use of any of these functions. In addition, DOS 3.x provides function *58H*(get/change memory allocation strategy), which controls how memory is allocated: first fit (the default in DOS 3.x and that used in DOS 2.x); last fit; and best fit.

Note that DOS (actually COMMAND.COM) uses these functions to allocate two memory blocks automatically when it loads and runs a transient program (using the DOS EXEC function *4BH*) and frees them upon termination. Any other memory allocated by an application using these functions must subsequently be freed or it will remain in memory. None of the memory allocated for a TSR is released automatically, of course.

A more detailed discussion of these memory-management functions would warrant a separate article. After using my program, all of them should make more sense (the official documentation for them is a little cryptic). Also, with the understanding of DOS memory allocation provided by my program and this article, DOS could be bypassed altogether for memory management, although that is not my intention.

Locating MCBs in Memory

Two of a program's MCBs can be found easily: one MCB is one paragraph above its PSP; the other is one paragraph above the environment block. The segment address of the copy of the environment is contained in the word at offset *2CH* in the PSP. (This is not true for the master copy of COMMAND.COM. Under DOS 2.0 – 3.2, the environment pointer for the master copy of COMMAND.COM has a dummy value of *0000*. DOS 3.3 corrects this oversight.) This doesn't help you find the MCBs lower in memory, though, because the MCBs are linked only in the forward direction. The trick is to locate the first

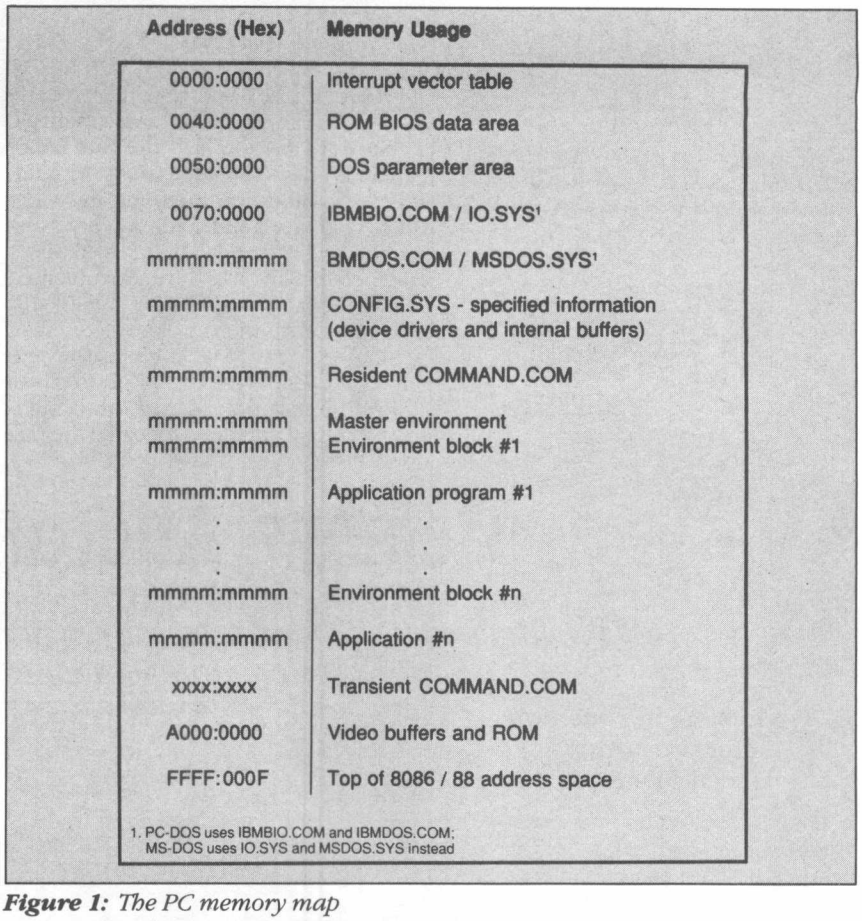


Figure 1: *The PC memory map*

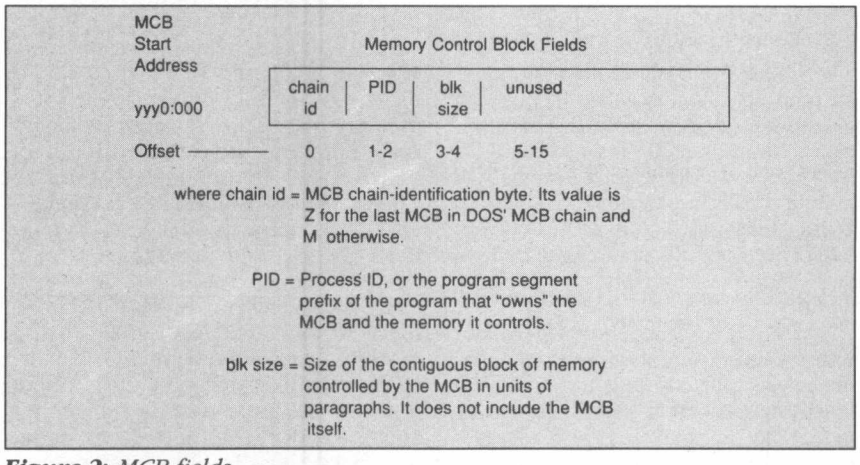


Figure 2: *MCB fields*

```
C> MCB
=============================================================================
MCB MCB   ID PID      MB   PAR- ENV   OWNER
NO. SEG             SIZE ID   BLK?
=============================================================================
01  0973 M  0008    8208 022B N     IBMDOS.COM/MSDOS.SYS
02  0B75 M  0B76    3376 0B76 N     COMMAND.COM COPY #1
03  0C49 M  0000      48 F000 N     FREE MEMORY CONTROL BLOCK
04  0C4D M  0B76     160 0B76 Y     COMMAND.COM COPY #1
05  0C58 M  0C5E      64 0B76 Y     C:\TURBOC\DEV\MCB.EXE
06  0C5D M  0C5E  71232 0B76 N     C:\TURBOC\DEV\MCB.EXE
07  1DC2 Z  0000 530384 F000 N     FREE MEMORY CONTROL BLOCK
=============================================================================
```

Example 1: *System booted with no CONFIG.SYS and no AUTOEXEC.BAT. (I was in the directory C:\TURBOC\DEV, where MCB.EXE resides on my system, for this and all subsequent examples.)*

MAPPING DOS

(continued from page 57)

MCB in memory —once you have done that, it is a simple matter to visit them all.

So how do you find the first MCB? There is no documented way. Any number of brute-force methods have occurred to me, none of them very satisfactory. Fortunately, there is a better method that involves the undocumented DOS *Invars* function *52H.* When this function call is complete, *ES:[BX-2]* points to a word that contains the segment address of the first MCB. Because all MCBs start on a paragraph boundary, that's all you need to locate the first one.

Obtaining Additional Information

Each MCB/MB pair is owned by a PID, which is nothing more than the segment address of the PSP of the program that owns the memory. From the PSP address, you can derive other useful information. First, you can find the location of the program's copy of the environment, as mentioned earlier. Another (undocumented) piece of information you can discover is the PID of the parent process, which is located at the word *PSP:16H.* In DOS 3.x only, you can also find the name of the owner program.

The PID owner name consists of the drive, path, and filename of the program associated with the PID. To find it, first locate the copy of the environment for any program other than the operating system files IBMDOS.COM and COMMAND.COM (these must be treated separately) using the word at *PID:2CH.* Each string in the environment is in ASCIIZ form (which ends in a NULL, the ASCII character with value 0). Search for the first double NULL (sometimes there is more than one).

If a word count of 0001 immediately follows the double NULL (indicating that only one more ASCIIZ string is left in the environment), then the owner

```
C>TYPE \CONFIG.SYS
device=c:\dos\ansi.sys
files=50
buffers=20
lastdrive=z

C>MCB

 ================================================================
 MCB MCB   ID PID      MB  PAR- ENV  OWNER
 NO. SEG             SIZE ID   BLK?
 ================================================================
 01  0973 M  0008  16352 022B N    IBMDOS.COM/MSDOS.SYS
 02  0D72 M  0D73   3376 0D73 N    COMMAND.COM COPY #1
 03  0E46 M  0000     48 F000 N    FREE MEMORY CONTROL BLOCK
 04  0E4A M  0D73    160 0D73 Y    COMMAND.COM COPY #1
 05  0E55 M  0E5B     64 0D73 Y    C:\TURBOC\DEV\MCB.EXE
 06  0E5A M  0E5B  71232 0D73 N    C:\TURBOC\DEV\MCB.EXE
 07  1FBF Z  0000 522240 F000 N    FREE MEMORY CONTROL BLOCK
 ================================================================
```

Example 2: *The PC rebooted with the CONFIG.SYS shown and no AUTOEXEC.BAT*

```
C>TYPE \AUTOEXEC.BAT
path c:\dos;c:\util;c:\turboc;c:\util\norton;c:\masm;c:\pe
prompt $p$g
mgc
astclock
subst u: c:\util
fastopen c:
cls

C>MCB
================================================================
MCB MCB   ID PID      MB  PAR- ENV  OWNER
NO. SEG             SIZE ID   BLK?
================================================================
01  0973 M  0008  16352 022B N    IBMDOS.COM/MSDOS.SYS
02  0D72 M  0D73   3376 0D73 N    COMMAND.COM COPY #1
03  0E46 M  0000     48 F000 N    FREE MEMORY CONTROL BLOCK
04  0E4A M  0D73    160 0D73 Y    COMMAND.COM COPY #1
05  0E55 M  0E5F    128 0D73 Y    C:\DOS\FASTOPEN.EXE
06  0E5E M  0E5F   2896 0D73 N    C:\DOS\FASTOPEN.EXE
07  0F14 M  0F1E    128 0D73 Y    C:\TURBOC\DEV\MCB.EXE
08  0F1D M  0F1E  71232 0D73 N    C:\TURBOC\DEV\MCB.EXE
09  2082 Z  0000 519120 F000 N    FREE MEMORY CONTROL BLOCK
================================================================
```

Example 3: *The system rebooted with the CONFIG.SYS file in Example 2 and an AUTOEXEC.BAT file with the contents shown*

of the program is given by the ASCIIZ string immediately following. If the end of the environment is reached and no such pattern is found, the owner is unknown. (This will always be the case for DOS 2.x.) Note that if a TSR has freed its environment block, the owner name thus found is likely to be inaccurate because the TSR no longer owns it and it may have been claimed by a different program (typically, the next TSR or transient program will claim it).

The MCB.C program provides insight as to how the DOS memory-management functions do their job and can show you exactly where all your memory has gone the next time you run out of it on a machine overloaded with TSRs

The MCB Program

The program MCB.C in Listing One is heavily commented, so I'll make only a few general comments here.

I used the small-memory model of Turbo C, Version 1.5, to compile and link MCB.C into MCB.EXE. The complications of doing so are manifested in the explicit declarations of some *far* and *huge* pointers, as explained in the comments.

I used no Turbo C library calls specific to either Turbo C itself or to the IBM PC, so other C compilers should have no problem compiling the program, and it should run on any MS-DOS machine, even one not compatible with the IBM PC. (I ran it without any difficulty on a DEC Rainbow under MS-DOS 2.11.)

The program itself is a straightforward implementation using the material discussed earlier in this article. The first MCB in memory is located using DOS function *52H*. The program then visits each one and computes and prints out information on each MCB/MB. See the large comment block at the end of function *prn_header* in Listing One for a detailed description of the fields printed for each MCB.

Examples 1 – 6, pages 57 – 63, contain the screen output of several illustrative examples. The conditions under which each was run are summarized in the captions. The effects of other DOS commands run are also shown when they are relevant to the discussion.

```
C>GRAPHICS

C>MCB
=================================================================
MCB MCB  ID PID     MB   PAR- ENV  OWNER
NO. SEG           SIZE ID   BLK?
=================================================================
01  0973 M  0008  16352 022B N    IBMDOS.COM/MSDOS.SYS
02  0D72 M  0D73   3376 0D73 N    COMMAND.COM COPY #1
03  0E46 M  0000     48 0F14 N    FREE MEMORY CONTROL BLOCK
04  0E4A M  0D73    160 0D73 Y    COMMAND.COM COPY #1
05  0E55 M  0E5F    128 0D73 Y    C:\DOS\FASTOPEN.EXE
06  0E5E M  0E5F   2896 0D73 N    C:\DOS\FASTOPEN.EXE
07  0F14 M  0F1E    128 0D73 Y    C:\DOS\GRAPHICS.COM
08  0F1D M  0F1E   2144 0D73 N    C:\DOS\GRAPHICS.COM
09  0FA4 M  0FAE    128 0D73 Y    C:\TURBOC\DEV\MCB.EXE
10  0FAD M  0FAE  71232 0D73 N    C:\TURBOC\DEV\MCB.EXE
11  2112 Z  0000 516816 0F14 N    FREE MEMORY CONTROL BLOCK
=================================================================
```

Example 4: *Running the DOS command GRAPHICS*


```
C>COMMAND

The IBM Personal Computer DOS
Version 3.30 (C)Copyright International Business Machines Corp 1981,1987
             (C)Copyright Microsoft Corp 1981, 1986

C>MCB

=================================================================
MCB MCB  ID PID     MB   PAR- ENV  OWNER
NO. SEG           SIZE ID   BLK?
=================================================================
01  0973 M  0008  16352 022B N    IBMDOS.COM/MSDOS.SYS
02  0D72 M  0D73   3376 0D73 N    COMMAND.COM COPY #1
03  0E46 M  0000     48 0F14 N    FREE MEMORY CONTROL BLOCK
04  0E4A M  0D73    160 0D73 Y    COMMAND.COM COPY #1
05  0E55 M  0E5F    128 0D73 Y    C:\DOS\FASTOPEN.EXE
06  0E5E M  0E5F   2896 0D73 N    C:\DOS\FASTOPEN.EXE
07  0F14 M  0F1E    128 0D73 Y    C:\DOS\GRAPHICS.COM
08  0F1D M  0F1E   2144 0D73 N    C:\DOS\GRAPHICS.COM
09  0FA4 M  0FAD    112 0FAD N    COMMAND.COM COPY #2
10  0FAC M  0FAD   3376 0FAD N    COMMAND.COM COPY #2
11  1080 M  0FAD    160 0FAD Y    COMMAND.COM COPY #2
12  108B M  1095    128 0FAD Y    C:\TURBOC\DEV\MCB.EXE
13  1094 M  1095  71232 0FAD N    C:\TURBOC\DEV\MCB.EXE
14  21F9 Z  0000 513120 0F14 N    FREE MEMORY CONTROL BLOCK
=================================================================
```

Example 5: *Installing a secondary copy of the command processor COMMAND.COM*

Example 1 illustrates the minimum number of MCB/MB pairs because no extra device drivers or TSRs have been installed. The curious free embedded MCB 03 seems to be a leftover from the DOS boot process —it seems strange that DOS should leave it there. Also note the use of memory by MCB.EXE and the final large free block. Many application programs claim all of the remaining memory for themselves when they run, and I was curious as to why MCB.EXE didn't. Because it was an .EXE file, it was possible that its .EXE header specified a smaller amount of memory. I checked the header and found it was set to claim as much free memory as was available. I finally traced the return of memory to DOS by examining the Turbo C start-up code that was executed prior to giving *main( )* control. Sure enough, this code called DOS function *4AH*. The parent address (*022BH*) of IBMDOS.COM in this example is also interesting.

In Example 2, the size of the MB controlled by MCB 01 and owned by IBMDOS.COM shows an increase of 8,144 bytes. The extra space was allocated during the boot process in response to the statements in CONFIG.SYS. These statements caused extra storage to be allocated for the installation of the device driver ANSI.SYS as well as that needed to carry out the *FILES=50* and *BUFFERS=20* commands. This block grows much larger when VDISK.SYS is installed. Try it.

In Example 3, the DOS 3.3 command

FASTOPEN is seen to be a memory-resident program that owns two blocks of memory—one for a copy of the environment and one for the resident code itself, which is the typical case.

Example 4 shows that the DOS *GRAPHICS* command is also a TSR and that it owns two more blocks of memory. Some TSR programs free their copy of the environment before executing the DOS *TSR* function. Such programs will not only have no environment block but also the owner of the remaining block holding the resident code may be incorrect because the owner string resides in the freed environment block. A TSR may have also allocated additional memory before exiting to DOS, in which case it would own additional memory blocks, which MCB.EXE would report.

In Example 5, the secondary copy of COMMAND.COM owns three blocks of memory for some reason. I don't know why, but it's interesting.

Example 6 shows that the space formerly occupied by the secondary COMMAND.COM has been freed upon execution of the *EXIT* command. You can install more than one secondary COMMAND.COM if desired, and you can remove each one in reverse order with an *EXIT* command.

In Example 7, *CHKDSK* reports 588,064 bytes free, whereas in Example 6 MCB shows 516,816 bytes free in the MB controlled by MCB 11. Let's reconcile the differences. The free memory reported by *CHKDSK* is equal to 516,816 (MCB 11) plus 16 plus 71,232 (MCB 10).

This makes sense. When *CHKDSK* runs, it has memory allocated to it (in fact all the remaining memory). When it reports the amount of free memory, it subtracts the memory needed to run itself but does not discount that needed for its copy of the environment, apparently reasoning that any program run will need to have such an environment block allocated. The extra 16 bytes in the equation above are needed to account for the paragraph of memory needed for the last MCB (11 here) itself. Also note that *CHKDSK* does not include the memory in any free memory embedded in the body of the allocation chain (such as MCB 03), even though such memory is available for use under the right conditions.

You'll also notice that the PC on which I ran this program seems to have only 652,288 bytes of total memory, which is 3,072 bytes less than the full 640K (655,360 bytes) of memory that is actually installed. This is an example of memory being hidden from DOS. My computer was an IBM PC XT equipped with a Paradise graphics card (which provides for monochrome text and CGA graphics on an IBM monochrome monitor). This video card also needs a memory-resident program (called MGC.COM, see the contents of the AUTOEXEC.BAT file in Example 3) installed in order for it to function. So why doesn't MGC.COM show up in one of the MBs?

It occupies the top 3,072 bytes of memory. After installing itself it modifies DOS' record of memory stored in word *40H:13H* and does a warm reboot, which reinitializes the computer without doing a memory check. The reason for this procedure is that MGC.COM must be in memory even when DOS is not being used (for example, for some games that need to be booted from a floppy). The MCB/MB scheme is used only by DOS and wouldn't be sufficient in such cases.

```
C>EXIT

C>MCB

=============================================================
MCB MCB  ID PID     MB  PAR- ENV  OWNER
NO. SEG           SIZE ID   BLK?
=============================================================
01  0973 M  0008  16352 022B N    IBMDOS.COM/MSDOS.SYS
02  0D72 M  0D73   3376 0D73 N    COMMAND.COM COPY #1
03  0E46 M  0000     48 0F14 N    FREE MEMORY CONTROL BLOCK
04  0E4A M  0D73    160 0D73 Y    COMMAND.COM COPY #1
05  0E55 M  0E5F    128 0D73 Y    C:\DOS\FASTOPEN.EXE
06  0E5E M  0E5F   2896 0D73 N    C:\DOS\FASTOPEN.EXE
07  0F14 M  0F1E    128 0D73 Y    C:\DOS\GRAPHICS.COM
08  0F1D M  0F1E   2144 0D73 N    C:\DOS\GRAPHICS.COM
09  0FA4 M  0FAE    128 0D73 Y    C:\TURBOC\DEV\MCB.EXE
10  0FAD M  0FAE  71232 0D73 N    C:\TURBOC\DEV\MCB.EXE
11  2112 Z  0000 516816 0F14 N    FREE MEMORY CONTROL BLOCK
=============================================================
```

Example 6: *The secondary COMMAND.COM remains in memory until a DOS EXIT command is issued*

MAPPING DOS

```
C>CHKDSK

 21309440 bytes total disk space
 53248 bytes in 2 hidden files
 59392 bytes in 28 directories
 9635840 bytes in 580 user files
 11560960 bytes available on disk

 652288 bytes total memory
 588064 bytes free
```

Example 7: *Running the DOS command CHKDSK to see how much free memory it reports and see how it compares to that reported by MCB.C*

Conclusion

The possible uses for this program are many. You could, for example, examine the effect of all of the DOS memory-allocation functions detail. You could possibly devise a scheme to remove any TSR from memory, not necessarily in reverse of the order installed, without creating a hole in memory. With the location of the master environment known, those who need to change the environment frequently (for example the DOS path) and find the use of *SET* tedious, could write a full-screen version of the DOS *SET* command. No doubt you will think of other uses.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Inserting Elements into a Basic Integer Array

Little assembler routines can make a big difference with your Basic programs

by Bruce Tonkin

If you do much serious programming in Basic, you ought to use some assembler routines. Assembly language is widely used to enhance the performance of C, but it has even greater importance in Basic.

Although Basic is the most popular programming language in use today, Basic compilers generally lack the kind of optimization more easily provided in "small" languages like C. Many times, that optimization isn't necessary; Basic has enough built-in commands (already highly optimized) to make typical business applications programs run at a very acceptable speed.

No language contains all possible commands, however, when a needed command is not available, the programmer must add it via a subroutine. Because most Basics don't optimize very well (or at all), such added features are often slow. Therefore, it's an even bigger advantage to use selected assembler routines in Basic than in languages such as C, where high optimization is relatively common. I'll present an example of this from my personal experience with Microsoft's QuickBasic 4.0.

QuickBasic 4.0 is typical of modern Basics. It has hundreds of commands and functions built in, but some of the ones you most often need seem to be lacking. In particular, there is no command to allow quick insertion of a value into an array.

In both C and Basic, it's possible to write a loop that will move individual elements of an array up one position to make room for a new element. In C, there's a good chance that the core of such a loop will be optimized by the compiler to an efficient MOVSB or MOVSW instruction. That's not so in QuickBasic 4.0 or in Microsoft's Basic 6.0.

Recently I wrote a sort/merge program. That's probably old stuff to most readers of this magazine, but this particular program turned out to be very interesting. The sort had two critical parts: a search routine (I used binary search to make things fast) and an index insertion routine. There wasn't very much I could do to speed up a binary search, but index insertion proved different.

The indices were in a single-dimensioned integer array that indexed an array of strings to be sorted. When a new string was added, its index was added to the pointer array by inserting the value at the position determined by the binary search. To insert a new element at position *K*, you might write something like this:

```
for i=lastelement+1 to k+1 step -1
    array(i)=array(i-1)
next i
array(k)=insertvalue
```

For speed, all variables will be integers, of course. Still, this short routine isn't very fast; the compiled code multiplies repeatedly to calculate each array position in turn before moving any data in the array. That's inefficient, because the loop is doing nothing more than moving the array elements (starting with the last element and moving down to element *K*) up two positions in memory; it's an ideal use for the REP MOVSW instruction, but neither the QB4 nor the Basic 6.0 compiler generate the obvious code.

Once I identified the bottleneck in my sort, I decided to write an assembler routine to perform the move. To keep the routine generally useful, I decided to write the routine so that it would move elements in either a static or dynamic array (via segmented addresses). I was very conservative with all the registers used, because I wanted the result to be as portable as possible for future (and other) Basic compilers. See Listing One, next page.

I was prepared for a modest speedup, but I was shocked when I prepared a simple benchmark. The performance increase was astounding! Table 1, this page, provides a summary.

If you're not familiar with assembler or just plain afraid of it, take a look at the listing of Iinsert (Listing Two, next page). I've commented it heavily so that you can see what it does.

If you do serious programming in Basic you could probably benefit from this and similar assembler tools. A rou-

Bruce Tonkin develops and sells software for TRS-80 and MS-DOS/PC-DOS computers. Although he writes mostly in Basic, Bruce also writes some software in C and assembly language. You may reach him at T.N.T. Software Inc. 34069 Hainesville Rd., Round Lake, IL 60073.

Benchmark Results	
QB4:	5765.63 seconds (3 tests)
Assembler:	41.48 seconds (4 tests)
Relative time, QB4/assembler:	139.0

(Times are for a Tandy 4000 with 16 MHz 80386)

Table 1: *Time to do 10,000 insertions at position 1 of a 10,000 element integer array*

tine to move long integers or floating-point numbers should be pretty easy to write, given this example. Usually, the results won't be as striking as they were in this case. Typical speed-ups in my assembler routines average about a factor of 10, but in critical parts of your programs a factor of even 2 or 3 is nothing to sneeze at.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings follow)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Listing One

The Test Program

```
defint a-z
rem $dynamic
dim a(10000)
for i = 1 to 10000: a(i) = i: next i
t! = timer
for i = 1 to 10000
   b = 9999
   call iinsert(seg a(1), b)
   a(1) = -i
next i
t1! = timer - t!
for i = 1 to 10000: print a(i); : next i
print
t! = timer
for i = 1 to 10
   for j = 10000 to 2 step -1: a(i) = a(i - 1): next j
   a(1) = i
next i
t2! = timer - t!
print t1!; "seconds for 10,000 assembler insertions."
print t2!; "seconds for 10 QB4 insertions."
END
```

End Listing One

Listing Two

Iinsert Program Listing

```
 .MODEL MEDIUM
 .CODE
 PUBLIC IINSERT
;Iinsert by Bruce W. Tonkin on 8-12-88 for QB 4.0 & MASM 5.0
;Iinsert will move, in descending order, elements of any
;1-dimensional integer array to allow a new element to be
;inserted.  It is called with:
;call iinsert (seg arg1,count)
;where arg1 is the element where the new value is to be inserted,
;and count is the integer number of elements to move.  e.g.:
;call iinsert(seg x%(5),y%)
;will move y% elements up one within the array, and you can then
;assign a new value to x%(5).  The value of x%(5) will be in
;x%(6) and x%(6) in x%(7), and so on for a count of y% elements.
;If the last element were x%(1000), then y% should be 1000-5=995,
;since the 999th element will go to the 1000th and elements 5
;through 999 will move up to elements 6 through 1000.  This
;routine works with either static or dynamic arrays, regardless
;of the array's location in memory--the DS register that
;indicates the current BASIC data segment is irrelevant.

Iinsert PROC
    push bp         ;save old BP
    mov  bp,sp      ;set framepointer to old stack
    push ds         ;save data segment--altered by routine
    push es         ;and extra segment--altered by routine
    push di         ;and destination index--altered by routine
    push si         ;and source index--altered by routine
    push ss         ;and stack segment--just to be safe
    mov  bx,[bp+6] ;address of the count parameter
    mov  cx,[bx]    ;count is now in CX
    shl  cx,1       ;multiply that by two for the moment
    les  di,[bp+8] ;segmented address of destination parameter
    lds  si,[bp+8] ;segmented address of source parameter, too
    add  di,cx      ;es:di points to end of destination
    add  si,cx      ;and ds:si points to end of source
    inc  di         ;move up one element for destination for an
    inc  di         ;integer insert--that's two bytes
    shr  cx,1       ;restore original cx value
;default destination is es:di, default source is ds:si for movsw
    std             ;set direction flag for decremented copy
    rep  movsw      ;move word at a time, since elements are words
    movsw           ;and move the original element up one, too
    pop  ss         ;restore saved registers
    pop  si
    pop  di
    pop  es
    pop  ds
    cld             ;clear direction flag--a well-mannered routine
    pop  bp         ;restore old base pointer
    ret  0ah        ;clear 10 bytes of parameters on return
Iinsert ENDP
    END
```

End Listings

PROLOG/V: PROLOG IN THE SMALLTALK ENVIRONMENT

A little known Smalltalk/V tool combines Smalltalk's object-oriented, class structured environment with Prolog's logical, declarative approach

by Gregory L. Lazarev

Recently, there has been a lot of interest in multilanguage software environments which provide a variety of tools that can be applied to different kinds of problems. Despite the fact that many applications can be implemented using one language, it seems worthwhile to combine the strong features of different languages into one environment. The subject of this article is the Prolog/Smalltalk environment, which is particularly intriguing because of the two powerful paradigms involved: logic programming (such as with Prolog) and object-oriented programming (as with Smalltalk). The environment examined is Digitalk's Smalltalk/V, which has an embedded version of Prolog, Prolog/V.[1]

Prolog

The power of the declarative programming language Prolog is well known. The source of this power is in describing knowledge rather than in prescribing a solution technique. A Prolog program consists of a set of non-ordered facts and rules (clauses) describing the knowledge required to solve a problem. Clauses are independent; each has its separate logical meaning. The "what" portion of the problem (that is, what is known and what is required) is expressed explicitly. Programming in Prolog is quite different from programming in conventional languages. Rather, it is closer to the development of a logical specification. The solutions are obtained primarily through the default inference mechanism that is a part of Prolog (the "how" portion is implicit).

This clear separation between logic and control results in many practical advantages of Prolog. The declarative style of programming is very natural, making the task of programming easier; programs are more concise and easier to maintain; Prolog provides an extension of the relational database technology; and the language is powerful and flexible enough to be used for all stages of software development, as well as for rapid prototyping.

These advantages of Prolog are supported by its features, many of which are unique to Prolog: both programs and data are represented by clauses; general pattern matching is available through unification; an automated inference mechanism with backtracking is provided; a non-deterministic style of programming is possible; use of recursive programs and structures can provide simplification; and many Prolog programs are invertible.

A problem usually can be presented as a set of declarative and procedural (imperative) tasks. Prolog is ideally suited for the declarative tasks. However, its applicability for procedural tasks based on side effects (such as I/O, windows manipulation, graphics interface, and so on) is more limited and less natural. Also, Prolog does not provide embedded facilities (classes, inheritance) for describing a problem's taxonomy and placing it within the existing environment. Therefore, the usability of Prolog can be substantially extended by merging it with other languages. These benefits work both ways: by extending the Prolog capabilities and by adding Prolog's inferential power to the other language. One example of such a mixed environment is the interface with the C language, supplied by most Prolog vendors. We consider here the Prolog/Smalltalk environment, which provides many features that cannot be found in other environments.

Gregory L. Lazarev is president of Applied Logic Programming Inc. He can be reached at 262 Tomkenn Rd., Philadelphia, PA 19151; 215-649-4740.

Object-Oriented Programming and Smalltalk

One problem with traditional software is the separation of data and procedures. Despite the fact that data and procedures are treated as independent, in reality they are not. Each procedure makes assumptions about the data it deals with. These assumptions are explicitly expressed in the procedure by using strong data types and through programming statements (such as the *case* statement). The separation of data and procedures often results in ad hoc, non-organized programming practice. Moreover, it prevents reuse of the procedures. Each procedure is tightly bound to the environment (application) and to the specific data structures it uses. Therefore, it is very difficult, if not impossible, to make such procedures general enough to be useful under different circumstances.[2]

Object-oriented programming (OOP) resolves this problem by binding data and procedures, known as methods, together as objects. The object's internal structure (data and methods) is hidden from the outside, which makes objects reusable. Computation is performed by sending a message to the object, and only the object itself, through one of its methods, determines the response to the message. In this way the external interface (through messages) defines what should be done, and the internal structure of the object defines how it should be done.

Everything in OOP is based on objects communicating through messages. For example, control structures are invoked by sending messages with blocks as arguments (that is, *ifFalse: aBlock, whileTrue: aBlock*, and so on). Objects are grouped into classes. Classes form a hierarchy; within the hierarchy each class inherits variables and methods from its superclass. For systems that cannot (or should not) be described using the traditional method of functional decomposition, the object-oriented approach is very natural and allows an easier transition from conceptualization to implementation.[3] Table 1, this page, provides a comparison of semantics and design methodologies for Prolog, Smalltalk, and conventional imperative languages, such as Fortran and C.

Smalltalk is the best-known example of OOP and is more than a language. It is an interactive, graphics-oriented programming environment based on a carefully chosen class organization. This window, menu-based environment integrates the language, the operating system, and such support tools as text editor, compiler, and debugger. In addition, the inspector lets you examine and edit objects, and browsers help you to navigate and manipulate within a maze of classes and methods.

Smalltalk/V includes numerous classes. Magnitude classes define objects that can be compared, measured, ordered, and counted (some examples of these classes are Number, Character, Date, and Time). Stream classes are used to access files, devices, and internal objects such as sequences of characters or other objects. Collections classes define arbitrary objects (examples include Bag, Set, Array, and String classes). Window classes include the Pane classes, used for the display in a designated area of the screen, and Dispatcher classes, used for processing keyboard and mouse inputs. The Dispatch Manager class schedules all the windows under its control. Graphic classes consist of four major classes: the Form class provides a two-dimensional view for a bitmap; the Point and Rectangle classes reference a point and a rectangular block contained in a Form; and the BitBlt (bit block transfer) class describes the basic bitmapped operation of moving a block of bits from one place to another. The latter also describes more complex operations such as conversion of characters into displayable bit patterns (class CharacterScanner), drawing lines from one place to another (class Pen), and animation (class Animation).

Prolog/V is fully embedded into the Smalltalk environment, therefore Smalltalk features, such as inheritance, are available

Object-oriented extensions are applied to several languages, including Prolog,[4] but they are beyond the scope of this article.

Prolog/V

The emphasis in the implementation of Prolog/V is on the tight integration of Prolog and Smalltalk. Prolog/V is fully embedded into the Smalltalk environment, therefore Smalltalk features, such as inheritance, are available. The class is defined first, then the Prolog clauses belonging to this class are entered in a way very similar to entering the Smalltalk methods. The only difference is that it is done through a Logic Browser window rather than through a Class Hierarchy Browser window, as it is for Smalltalk methods. (The Logic Browser has a Prolog compiler, the Class Hierarchy Browser has a Smalltalk compiler.) All clauses with the same name, that is with the same predicate name in the clause head, are grouped together and compiled into one Smalltalk method with one argument. A class Logic (a subclass of Object) and its subclass Prolog provide the methods for Prolog execution and built-in predicates, respectively. Therefore, in order to inherit these properties, any Prolog program should be placed into one of the subclasses of the Prolog class.

You can call Prolog from Smalltalk and vice versa. From Smalltalk, you can invoke Prolog's question. The standard Prolog question *?- a1, a2, ..., aN* is replaced in Prolog/V by the *receiver :? a1, a2, ..., aN*. Here the *receiver* is an instance of some arbitrary class. The clauses matching the goals *a1, a2, ..., aN* may either belong to this class or be inherited from its superclasses. After calculating the answers, Prolog returns them as a Smalltalk array (or as a *nil* if there are no answers) that can be processed further by Smalltalk if necessary.

On the Prolog side, the predicate *is* used to send a Smalltalk message. It provides a convenient way to support side-effects in Prolog/V. The predicate *is* as in *is (var, smalltalkExpr)* unifies a Prolog variable *var* with the Smalltalk expression *smalltalkExpr*. It replaces the standard *is* predicate in Prolog that unifies a variable with the Prolog expression. Therefore, in Prolog/V the arithmetic is performed by sending a

	Semantics	Methodology
Prolog	Declarative	Functional decomposition
Smalltalk	Procedural	Object-oriented design
Imperative languages	Procedural	Functional decomposition

Table 1: *Comparison of semantics and design methodologies for Prolog, Smalltalk, and conventional imperative languages*

SMALLTALK/PROLOG

(continued from page 69)

Smalltalk message as in *is(x,2 + 3 * 4)*.

Two other examples of using the *is* predicate are:

```
is( len, len0 value + 1)
```

and

```
is( _, pen
    place: 5 @ 5;
                goto: 35 @ 25)
```

In the first example *len0* is a Prolog variable. The Prolog variable can be used in a Smalltalk expression, but the *value* message should be sent to such a variable first. In the second example, only the side-effect (drawing a line) is relevant.

Prolog/V syntax is quite close to the standard Edinburgh syntax.[5] In addition to the two previously described differences (the question and the *is* predicate), there are other major differences. Prolog/V variables start either with a lower case letter (local variables) or with an upper case letter (global variables). Atoms are preceded with #, characters are preceded by *$*, strings must be surrounded by single quotes, and no underscore is allowed in names (the anonymous variable _ is an exception). All structures, even structures without arguments, must have a pair of parentheses as in *pred1(x, y)*, *pred2( )*. Some predicates are not implemented in Prolog/V; in particular, side-effect predicates (such as *get*, *put*, *see*, *seen*, etc.) are supported through the Smalltalk messages. Others may have different semantics. For example, the *consult* predicate provides communication among parallel classes.

To demonstrate the advantages provided by the Prolog/Smalltalk integration, consider an example of a program for transferring blocks. The next section presents a core program written in standard Prolog (Arity/Prolog is used). The extensions provided by Prolog/V (multi-pane windows, synchronization among panes, customized menu, graphics interface) follow.

Blocks Example in Standard Prolog

The blocks example used here is taken from Lazarev.[6] Consider a tabletop with a pile of blocks on it. Blocks may be directly on the table or on each other. The problem is to create a plan which moves the blocks from a given starting configuration to a given final configuration. Blocks are moved one at a time if the following conditions are satisfied: the top of a block to be moved must be clear; if a block is moved to the top of another block, then the top of the second block also must be clear.

The resulting program is shown in Listing One, page 98. It is based on the state-space search using the generic recursive procedure path. The *move* procedure is problem-specific.

The path has three arguments, the start state (*State*), the goal state (*Goal*), and the history of visited states (*Hist*). If the *Goal* is reached, then the solution is printed using the procedure *printpath*. Otherwise, in order to find a path from the *State* to the *Goal*, the following steps should be taken: find *move* from a *State* to an intermediate state *Interm*; check that *Interm* state is not on the list of already visited states, *Hist*; and find a path from *Interm* to a *Goal* with the list of visited states enlarged by *Interm*. The top goal *go* calls *path* with *[Start]* as the initialized list of visited states, and the *!* predicate limits search to the first solution.

The blocks problem is represented using the list data structure:

```
[on( a,A), on( b,B), ..., on( z,Z) ],
```

where a, b,...z are blocks, and A, B,...,Z are either blocks or the table *t*.

Two *move* clauses describe the transition from a *State1* to a *State2*. The first clause chooses some block (using a *member* predicate), checks if it is clear, and places the chosen block on the table. The resulting *State2* is formed by substituting *t* for the original block's position *Y*. The move of a block al-

(continued from page 70)

ready located on the table is forbidden. The second *move* clause is similar to the first but places the block on top of another block rather than on a table. The second block must be different from the first and clear. As a result, the list's element *on(X,Y)* is replaced by *on(X,Z)* where *X* is the block moved and *Y* and *Z* are its initial and final positions. The procedure *clear* specifies that a block *X* is clear if there is nothing on top of it.

For simplicity, let's limit ourselves to four blocks. The predicates *blocks1* and *blocks2* describe the three-block transfer; the predicate *blocks3* describes the four-block transfer. For example, consider the three blocks transformation corresponding to the goal *blocks2* (see Figure 1, this page).

The solution is presented as:

```
?- blocks2.
   [ on(a,t), on(b,t), on(c,a) ]
   [ on(a,t), on(b,t), on(c,t) ]
   [ on(a,t), on(b,a), on(c,t) ]
   [ on(a,t), on(b,c), on(c,t) ]
   [ on(a,b), on(b,c), on(c,t) ].
```

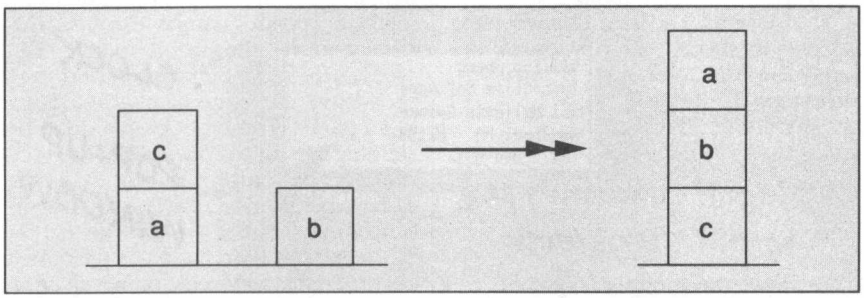


Figure 1: *Three blocks problem*

Blocks Example In Prolog/V

The core of the blocks program written in Prolog/V is the standard Prolog program from the previous section. The difference is an advanced user inter-

Smalltalk Terminology

Everything in Smalltalk centers around objects. Objects belong to classes, which, by forming a hierarchy, define inheritance. Each class is characterized by name and has instance and class variables associated with it. Each variable refers to the object whose pointer it contains. Assignment expression assigns the object to the variable (as in *array :- #(a b c)*). Instance variables (named and indexed) belong to the instance of the class and exist for the object's lifetime. Class variables are shared between all instances of the same class and exist until explicitly deleted. Class variables are a subset of shared variables that start with an uppercase letter and are shared by many objects.

Computation is defined as a process of changing instance variables of existing objects, creating new objects or destroying them. All computations are performed by messages. The only way to access the object's data in Smalltalk is to send a message to the object. A message is composed of three parts: a receiver of the message, the message selector (specifying the action to be performed), and the message arguments. A message always returns a single object as its result. The receiver of the message can be any object, that is an instance of the class or a class itself. The mechanism of the message passing provides external interfaces in Smalltalk. It takes the place of a procedure call in conventional languages.

There are three kinds of messages: unary messages, binary messages, and keyword messages. Unary messages are messages with no arguments. For example, *Pen new 'this is a string' size.* The first message with the selector *new* is the class message. It returns the new instance of the class *Pen*. The second message with selector *size* is the message to the instance of the class *String*. It returns the size of the string. Binary messages are messages with one argument and one or two special characters as selector. For example, *3 + 4* or *'abc', 'def'*. The first message with the selector + returns the object 7, the second message concatenates two strings and returns the string *'abcdef'*. Keyword messages are messages with one or more arguments. For example, *#(a b c b d b a) occurrencesOf: #b* or *#(a b c) at:2 put:$d*. The first message with the selector *occurrencesOf:* returns the number of b-occurrences in the array, i.e. 3. The second message with the selector *at:put:* returns *$d*, but it also replaces *b* in the array *#(a b c)* with *$d*.

Normally, these messages are combined together into expressions with predefined rules of precedence. For example, in the expression *3 factorial + 7*, the unary message *factorial* is evaluated first, the binary message + follows. The result is 13.

In response to a message received by the object, a matching method is executed. Methods are procedures associated with the class. They keep the internals of the object implementation invisible from the outside. There are two kinds of methods: class methods, responding to messages sent to the class, and instance methods, responding to messages sent to instances of the class. A method consists of the message pattern (the selector and its arguments) that is matched with the invoking message, temporary internal variables, and a series of expressions separated by periods. The execution of these expressions defines the message result. Consider, for example, the instance method *includes* from the class *IndexedCollection*.

```
includes: anObject
    "Answer true if the receiver
    contains an element equal to
    anObject, else answer false."

  | index |
  index := self size + 1.
  [(index := index - 1) > 0]
    whileTrue: [
      anObject = (self at: index)
        ifTrue: [^true]].
  ^false
```

It has the selector *includes:* with one argument, and one temporary variable —*index*. The caret (^) symbol determines the value to be returned. Each element of the receiver is compared with *anObject* until either the match is found (the result is *true*) or not found (the result is *false*). This method is matched against the following messages

```
#(a b c) includes: #b    (returns true)
#(a b c) includes: #d    (returns false)
```

A class is characterized by its name, variables (class and instance), and methods (class and instance). Classes form a hierarchy with a class Object as the root. A class inherits from its superclasses instance and class variables and methods. This inheritance allows references to the variables and methods not defined within a class. Search for variables and methods starts within a receiver's class and, if it fails, the search continues in its superclass, the superclass of superclass, and so on. —G.L.

(continued from page 72)

face provided by Prolog/V. An interface of this kind is difficult to implement in other environments.

The blocks example in Prolog/V is represented by the class Blocks and its subclass BlocksPro. Their position within the hierarchy of classes is shown in Figure 2, below.

The class Library (a subclass of the class Prolog), has many useful predicates (such as *append, member, findall,* etc.) that are not among the built-in predicates of the class Prolog. The class Library is not shown here. The class Blocks, with its Smalltalk methods, is shown in Listing Two, page 98. The class BlocksPro (Listing Three, page 100) is the subclass of Blocks. All of its methods are Prolog clauses. BlocksPro doesn't have its own instance variables; it inherits instance variables and methods from the class Block.

The program is invoked by sending the message:

```
BlocksPro new openOn
```

The *openOn* method, inherited from the class Blocks, creates a multi-pane window BLOCKS. There are four panes interacting with each other —instances of the class SubPane. The list pane at the screen's top left corner displays the list of choices. The text pane at the screen's top right corner provides a brief description of the application and has a customized menu with two options —*do* and *help*. The text pane in the middle of the screen displays help messages and the analytically represented output (the output stream, defined by the instance variable *replyStream*, is determined by sending the *dispatcher* message to the object associated with the temporary variable *replyPane*). The graph pane at the bottom of the screen provides a graphic drawing of the moving blocks.

The position and size of each pane relative to the whole window is defined by the message *framingRatio:extent:*.

Prolog/V combines the Smalltalk object-oriented, class structured environment with the logical, declarative approach in specifying objects' behavior

A message *name:* is sent to the instance of the class SubPane. It initializes a pane when the window is first open. There are four initialization methods: *choices*, *input*, *reply*, and *graph:*. The method *choices* returns a list of available alternatives. The method *input* returns the text associated with the choice. The choice is defined by the instance variable *selectedChoice*. The method *reply* clears the output pane. Finally, the method *graph:* returns the white form of the specified size and initializes the global variables.

The class SubPane also provides the capability for synchronizing panes through the method *change:*. The message *change: #choice:* is sent to the instance of the class ListPane. The method *choice:* assigns the *selectedChoice* to the chosen alternative (i.e., *blocks1*, *blocks2*, or *blocks3*) and then broadcasts the value of *selectedChoice* to all panes. The broadcast is done by sending messages *changed:* to the controlling application (in our case, the instance of BlocksPro). The arguments of *changed:* messages are the names of the initialization methods described above. In this way, a list pane selection is synchronized with the contents of other panes. The initial screen display is shown in Figure 3, below.

After the choice is made, the next step is to access the text pane menu through the method *doBlocksMenu*. This menu gives two options *do* and *help*, defined by the methods *doBlocks* and *help*, respectively. (Notice that all methods described so far are inherited by the class BlocksPro from its superclasses, particularly from the class Blocks.) The *help* method writes the text of the help message into the output stream. The *doBlocks* method provides the main program function. It invokes one of the Prolog goals (*blocks1( )*, *blocks2( )*, or *blocks3( )*), depending on the value of *selectedChoice*. All Prolog clauses belong to the class BlocksPro. Only new clauses or clauses different from the standard Prolog program (Listing One) are described here.

The predicate *go/2* consists of three subgoals *init/1*, *path/3*, and the trivial subgoal described by the method *finish*. The functions of *init/1* predicate are to initialize the animation, to initialize the Smalltalk blocks data structure, and to draw the initial blocks configuration. The first two functions are described by the method *initialize1*. To invoke this method, as well as any other Smalltalk method from Prolog/V, a Smalltalk message should be sent using the predicate *is*, as in

```
is( _, self initialize1).
```

The method *initAnimation* adds four objects with different names and colors to the instance variable *animator*. Each object is represented by the array of forms simulating its continuous movement on the screen. When the object moves, the old image is erased, and the new image, taken from the array of forms, is displayed at the new position.

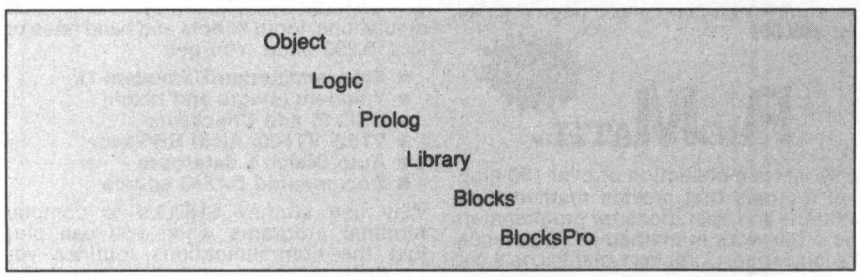


Figure 2: *Class hierarchy for the blocks example*

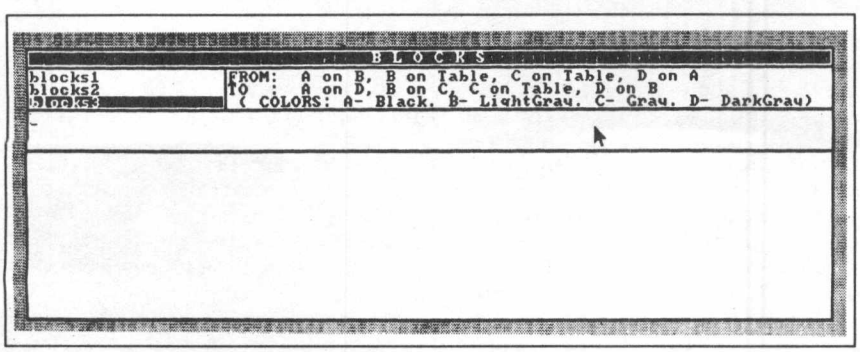


Figure 3: *The initial screen*

(continued from page 74)

The array of forms in the blocks application consists of one element only — the form White.

The existing Prolog blocks data structure is not adequate to support graphics of moving blocks. Therefore, the extra Smalltalk data structure, described by the instance variable *position*, is provided. (The other choice is to include the graphics extension within the existing Prolog data structure.) The variable *position* is represented as an array with the number of elements equal to the number of participating blocks. Each element is the instance of the class

Prolog Terminology

The syntax of Prolog is simple and is based on the concept of objects (not to be confused with the Smalltalk objects) and their relations.

The lowest level building blocks for Prolog programs are "terms." There are three classes of terms: constants, variables, and structures. Constants are the names of defined objects or specific relationships between them. Two main categories of constants in Prolog are atoms which normally begin with a lowercase letter (for example, *find_next*) and numbers. A constant represents a defined object. A variable stands for a yet undefined object that will be substituted once by an object in the course of computation (a process known as "instantiation"). Variables start with capital letters or the underscore character (for example, *XY, _Age*). The third class of terms is known as structures. A structure is described as: *f(t1, ..., tn)* where *f* is called a functor and *t1, ..., tn* are called components of the structure. A functor is characterized by its name *f*, which is an atom, and by the number of components *n* —its arity. It is denoted as *f/n*. A functor in Prolog cannot be a variable. This reflects a limitation of the first-order logic on which Prolog is based. A component of a structure may be any term, i.e., a constant, variable, or another structure. This allows the existence of nested structures.

Structures are fundamental in Prolog. A structure may either describe a single complex object built from other objects, as in: *person_birthday (Name, date (Month, Day, Year))* or it may describe a predicate which expresses a relationship among objects, as in the predicate *likes/2: likes(X, steve)*.

A Prolog program consists of clauses. Each clause is represented through predicates *A, B1, ..., Bn* in the form: *A :- B1, ..., Bk, ..., Bn* where *A* is the clause's head and *B1, ..., Bk, ..., Bn* constitute the clause's body. Declaratively, the clause *A :- B1, ..., Bk, ..., Bn* stands for: *A* is true if all B_i's are true.

The predicates *A* and B_i are also known as goals (or subgoals). As a predicate, a goal is either an atom or a structure. There is only one predicate in the clause's head. Commas between goals stand for conjunction (i.e., logical AND), but disjunction (logical OR) is also supported by separating goals with semicolons. The symbol *:-* is understood as *if*. Only two possible values are associated with each predicate: true or false. All variables appearing in clauses are read as "for all," i.e., universally quantified. The scope of a variable is limited to the clause in which it appears.

Prolog programs are described by facts, rules, and questions. All of them are all specific cases of clauses. A fact is a clause without body (that is, no conditions); rules have both a head and a body; questions are clauses without a head. The following notation is used for questions *?- B1, B2, ..., Bn* instead of the more formal *:- B1, B2, ..., Bn*.

The most fundamental feature of Prolog, which differentiates it from conventional languages, is its declarative character. Ideally, writing a program in Prolog means expressing *what* is known (that is, to represent facts and rules in the most natural way), as well as specifying a problem by formulating a question. Prolog will do the rest, i.e. it will produce answers by manipulating the supplied facts, rules, and question. The ability to support the computation process automatically is based on the fundamental principles of unification and resolution. Strictly speaking, the above scenario is valid for pure Prolog only. In real Prolog, augmented with many built-in predicates, it is approximately true.

Rules in Prolog have two interpretations. Besides the declarative interpretation, the rule *A :- B1, B2, ..., Bk, ..., Bn* has the following procedural interpretation: To satisfy goal *A*, first satisfy subgoal *B1*, then satisfy subgoal *B2, ...*, then satisfy subgoal *Bk, ...*, and last satisfy subgoal *Bn*.

This interpretation not only specifies which subgoals should be satisfied but also provides procedural information by answering how (i.e., in what order) the computation will be done. The procedural meaning of a program, in contrast to its declarative meaning, generally depends on the order of goals in the body of each clause as well as on the order of clauses within the program.

Computation in Prolog is based on a standard strategy augmented with backtracking. The standard strategy is defined as sequential, depth first processing with goals in the clause body invoked from left to right and each goal unified with the head of the matched clause chosen according to textual order within a procedure (a procedure is a set of clauses with the same head).

Unification is the process of finding a set of bindings for variables so that the terms match. It is responsible for the answer extraction. Two Prolog terms match if they are identical or if variables in those terms can be instantiated (i.e., bound) in such a way that they become identical.

If one of the goals fails, then the attempt to resatisfy the last satisfied goal takes place —a process known as "backtracking." All variables instantiated with the last satisfied goal are uninstantiated. This is the way the "undo" operation is performed in Prolog.

As an example, consider a program that defines whether a particular element is part of a list. (A list is an ordered sequence of elements with any length. Head is the first element of a list. Tail is the original list without its first element. The notation *[H | T]* is used with *H* as a head, and *T* as a tail.) Declaratively, the program is described as: X is a member of a list if it is identical to the head of the list or X is a member of a list if it is a member of the tail of the list.

The program is written in Prolog using the predicate *member(X, List)*:

```
member( X, [X|_]).
member( X, [_|T]) :- member( X, T).
```

Consider the goal *?- member(X, [a,b,c])*. The three solutions are: *X = a; X = b;* or *X = c*.

The first solution (*X = a*) is produced by unification of the goal with the first *member* clause. The system backtracks looking for other solutions. By unifying with the second *member* clause, the goal *?- member(X, [b,c])* is invoked. The process repeats itself (starting with the first *member* clause) and results in two more answers. —G.L.

OrderedCollection; it is initialized by the method *initPosition* to an empty collection.

Finally, the initial drawing is handled through the recursive predicate *initDraw/1*. It describes either the placing of block *x* on the table (predicate *addDraw(x))*, or the placing of block *x* on top of block *z* (predicate *addDraw (x, z)*). The actual work is performed by the Smalltalk methods *add:onTbl:* (called from *addDraw/1*) and *add:onBl:* (called from *addDraw/2*). Because the initial drawing is done in specific order (from the table up), the predicate *arrange/2* is introduced. It transforms the starting Prolog blocks data structure into the "drawing order" structure. For example, the structure *[on(a,b), on(b,t), on(c,t), on(d,a)]* is transformed into the structure *[on(c,t), on(b,t), on(a,b), on(d,a)]*. The screen display at the start of graphics simulation is shown in Figure 4, this page. Both menu options (*do* and *help*) were chosen.

The main procedure *path/3* is a modified version of the same procedure from the standard Prolog program. The first *path* clause is invoked when the goal is achieved. The text output pane is cleared, and all steps needed to transform the initial configuration into the final configuration are written into the output stream. The second *path* clause supports the graphics drawing by calling the procedure *moveDrawInOut/3*. The three parameters of the predicate *moveDrawInOut* describe the block's name, the place it is moving from, and the place it is moving to. (In order to obtain these parameters explicitly, the predicate *move/2* from the standard Prolog program was converted into the predicate *move/5*.) The procedure *moveDrawInOut/3* has two clauses. The first clause describes the move from *placeFrom* to *placeTo*. The second clause describes the reverse process — the move from *placeTo* to *placeFrom*. This clause is used on backtracking to undo the graphics effects from the last move. The procedure *moveDraw/3* describes the same two cases as the *move/5* procedure, i.e., placing a block on the table and placing a block on the top of the another block. It deals with updating the data structure *position* and also moves blocks on the screen from one place to another.

The actual work is done by the Smalltalk methods *remove:*, *add:onTbl:*, and *add:onBl:*. The method *remove: block* finds in the array *position* the ordered collection that includes *block*, and then removes *block* from there. There are no graphics involved. The methods *add: block onTbl: col* and *add: block1 onBl: block2* add the block after the last entry in the ordered collection and then move the block on the screen by sending the message *tell:place:* to the *animator* object. The only difference between these two methods is the mechanism of finding the proper ordered collection in the array *position*.

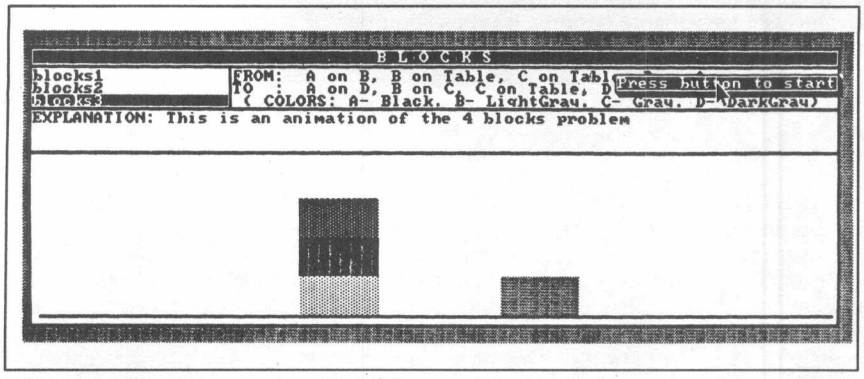


Figure 4: *The screen at the start of the graphics simulation*

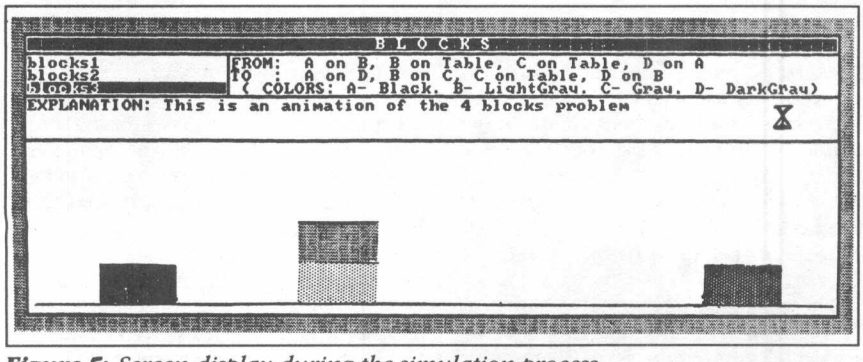


Figure 5: *Screen display during the simulation process*

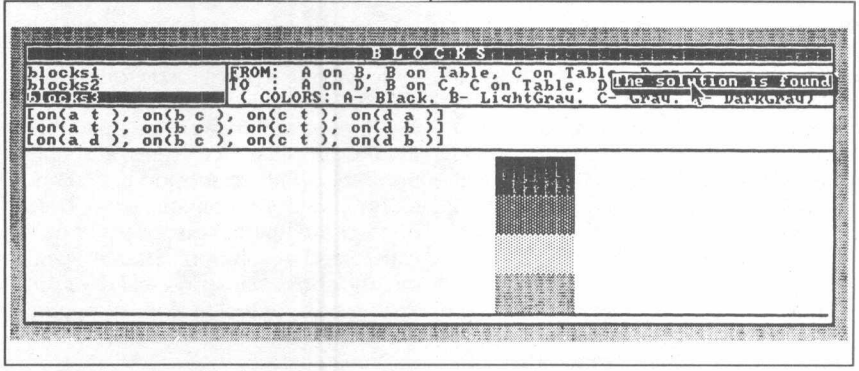


Figure 6: *Screen display after simulation*

Screen displays in the process of simulation and after program completion are shown in Figure 5, page 78, and Figure 6, this page.

Conclusions

Overall, the Prolog/Smalltalk environment and its excellent implementation by Digitalk provides an interesting example of multi-language systems. (The performance modeling tool based on Prolog/V is described in Pazirandeh and Becker.)[7] Many existing Prolog programs can easily be incorporated into this environment. The blocks application, described here, is an example of such a program.

The technique discussed above may also have a strong impact on software engineering practice. Prolog/V combines the Smalltalk object-oriented, class structured environment with the logical, declarative approach in specifying objects' behavior. The declarative approach advocated by Prolog is often more suitable than the traditional, procedural approach used in "pure" Smalltalk.

References

1. Digitalk *Smalltalk/V. Object-Oriented Programming System*, (Los Angeles, CA: Digital Inc., 1987).
2. B.J. Cox, *Object-Oriented Programming. An Evolutionary Approach*, (Reading, MA: Addison-Wesley, 1986).
3. B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987, 50 – 63.
4. E.P. Stabler, Jr., "Object-Oriented Programming in Prolog," *AI Expert*, Oct. 1986, 46 – 57.
5. W.F Clocksin and C.S. Mellish, *Programming in Prolog*, (Berlin: Springer-Verlag, 1984).
6. G.L. Lazarev, *Why Prolog? Justifying Logic Programming for Practical Applications*, (Englewood Cliffs, NJ: Prentice-Hall, 1989).
7. M. Pazirandeh and J. Becker, "Object Oriented Performance Models with Knowledge-based Diagnostics," *Proceedings of the 1987 Winter Simulation Conference*, (A. Thesen, H. Grant, W. David Kelton (eds.)), 1987.

Acknowledgment

I would like to thank Susan Bartley and Kenneth Krigelman for their valuable comments.

DDJ

(Listings begin on page 98.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

GRAPHIC COMPRESSION

Listing One *(Text begins on page 42.)*

```
/*              Bitmap compaction program.                              */
/* Converts bitmaps in 82786 graphics memory to Graphics processor      */
/* instructions. Simulates run length encoding, causing data            */
/* compression for most bitmaps. Compression of an 8 bits per pixel     */
/* down to 3 bits per pixel is not uncommon.                            */
/*        Created by Victor J. Duvanenko                                */
/* Usage:          compact output_file_name                             */
/* Input:                                                               */
/*       An 82786 eight bits per pixel bitmap at address of 0x10000     */
/*       (this can be easily changed). The bitmap is 640 pixels         */
/*       wide and 480 pixels heigh (this can also be changed). This     */
/*       was done for the simplicity of this example.                   */
/* Output:                                                              */
/*       Binary file in the following format.                           */
/*       a) Header                                                      */
/*          1) type ( 0 - GP instructions, 1 - bitmap data ).           */
/*          2) 32 bit address (lets the loader know where to place      */
/*             the data in graphics/82786 memory). If equal to          */
/*             0xffffffff then the loader can place data anywhere.      */
/*          3) number of bytes to load.                                 */
/*       b) Data                                                        */
/*          1) define color instruction                                 */
/*          2) scan line instruction                                    */
/*          3) link instruction (link around the array)                 */
/*          4) scan line array                                          */
/*       Caution: The loader must add a 'stop' instruction to the       */
/*                data (a NOP instruction with GCL bit set). See        */
/*                loader source code for example.                       */
/*                                                                      */
/* This program was written for the XENIX environment, but can be       */
/* easily and quickly ported to the PC. Just concentrate on the         */
/* ideas and not on the implementation specifics. I'll try to point     */
/* out XENIX specific sections.                                         */

#include<stdio.h>
#include<fcntl.h>
#include<sys/param.h>
#include "/usr/tls786/h/tls786.h"
#include "/usr/tls786/h/tvl786.h"

#define MAX_NUM_COLORS  256 /* maximum number of colors   */
#define GP_BUFF_SIZE   8192 /* GP instruction buffer size */
#define BITMAP_WIDTH    640
#define TRUE              1
#define FALSE             0
#define OK             TRUE /* return status for procedures          */
#define PMODE          0644 /* protection mode of the output file    */
#define MOVSW_ON       TRUE /* enable 'move strings' in XENIX driver */
#define DEBUG         FALSE /* top debug level - a fun one to turn on*/
#define DEBUG_1       FALSE /* next debug level deeper               */
#define DEBUG_2       FALSE /* everything you'd ever care to trace   */

/* Global data buffers - used by several routines */
unsigned int   gp_buff[ GP_BUFF_SIZE ];  /* GP instruction buffer    */
unsigned int   buff[ GP_BUFF_SIZE ];     /* temporary storage buffer */
unsigned char  line_buff[ BITMAP_WIDTH ];/* line buffer              */
            /* line_buff should ideally be dynamically alocated to   */
            /* allow any bitmap width. Left as an exercise for       */
            /* a C hacker.                                           */

int  p6handle;              /* XENIX file descriptor for the 82786   */
                            /* memory                                */
int  bm_height  = 480;      /* bitmap height                         */
int  bm_width   = 640;      /* bitmap width                          */
long bm_address = 0x10000L; /* bitmap address                        */

/* Description of each color, histogram, plus additional fields for  */
/* added dramatic performance improvement (defines a window of color  */
/* existance). Enable DEBUG to see time savings that result from      */
/* this technique.                                                    */
struct  color_struct
{
  long  count;      /* number of times a color appears in the bitmap */
  int   begin_y,    /* scan line  where a color first appears        */
        end_y,      /* scan line  where a color last  appears        */
        begin_x,    /* x position where a color first appears        */
        end_x;      /* x position where a color last  appears        */
};
typedef struct  color_struct  color_t;

/* Array containing color information about a bitmap under analysis  */
color_t colors[ MAX_NUM_COLORS ];

/*--------------------------------------------------------------------*/
/*          The body of the data compression program.                 */
/*--------------------------------------------------------------------*/
main(argc,argv)
int argc;
char *argv[];
{
    register i, index, j, num_colors;
    int  n, value, x, y;
    int  f1, f2,                        /* file descriptors */
         buff_overflow;

     /* the following variables are for debug purposes only */
     long     average_begin_y, average_end_y;
     long     average_begin_x, average_end_x;
     float     percentage_y, percentage_x;
     color_t *clr_p;

     /* XENIX needed structures needed for movsw section of           */
     /* the driver                                                    */
     union {
             struct tvl_cntrl brddata;
               byte   rawdata[ sizeof( struct tvl_cntrl ) ];
             struct io_data      rdata;
     } regdata;
```

```
  /* Check the command line for proper syntax, a little */
  if (argc < 2)
  {
    fprintf( stderr,"Usage is: %s output_file_name\n", argv[0] );
    exit(1);
  }

  /* Open the file where compacted bitmap will be placed. If it      */
  /* doesn't exist create it.                                        */
  if (( f2 = open( argv[1], O_CREAT | O_WRONLY, PMODE ))  == -1 )
  {
     fprintf( stderr, "%s: Can't create %s.\n", argv[0], argv[1] );
     exit(1);
  }

  /* XENIX specific functions to allow one to treat 82786 graphics   */
  /* memory as a file - a file descriptor gets passed to every       */
  /* routine that talks to the 82786. This allows a very flexible    */
  /* multiple 82786 environment.                                     */
  p6handle = open786( MASTER_786 );
  if ( p6handle == NULL )     {
     fprintf( stderr, "%s: Can't access 82786.\n", argv[0] );
     exit(1);
  }

#if MOVSW_ON
  /* XENIX specific - enable movsw in the 82786 driver */
  ioctl( p6handle, TEST_SFLAG, &regdata );
  if ( regdata.rdata.regval == FALSE )
     ioctl( p6handle, SET_SFLAG, &value );
#endif

  /* Find all unique colors in the bitmap file. */
  num_colors = find_all_colors( colors, bm_height );

#if DEBUG
  /* histogram and performance improvement information of the color  */
  /* existance window technique.                                     */
  printf( "num_colors = %d\n", num_colors );
  average_begin_y = average_end_y = 0L;
  average_begin_x = average_end_x = 0L;
  for( i = 0; i < MAX_NUM_COLORS; i++ )
  {
     clr_p = &colors[i]; /* for denser and cleaner notation purposes */
     printf( "c %4d %7ld  b_y%4d  e_y%4d", i, clr_p->count,
                                  clr_p->begin_y, clr_p->end_y );
     printf( "  b_x%5d  e_x%5d\n", clr_p->begin_x, clr_p->end_x );

     /* average only the existing colors */
     if ( clr_p->count != 0 )
     {
          average_begin_y += (long)clr_p->begin_y;
          average_end_y   += (long)clr_p->end_y;
          average_begin_x += (long)clr_p->begin_x;
          average_end_x   += (long)clr_p->end_x;
     }
  }
  printf( "\n" );
  average_begin_y /= (long)num_colors;
  average_end_y   /= (long)num_colors;
  printf( "average Y begin = %ld\t\taverage Y end = %ld\n",
                                  average_begin_y, average_end_y   );
  percentage_y  = ((float)( average_begin_y ) / bm_height * 100 );
  percentage_y += ((float)((long)( bm_height ) - average_end_y ) /
                  bm_height * 100 );
  printf( "percentage Y savings = %2.2f\n", percentage_y );

  average_begin_x /= (long)num_colors;
  average_end_x   /= (long)num_colors;
  printf( "average X begin = %ld\t\taverage X end = %ld\n",
                                  average_begin_x, average_end_x   );
  percentage_x  = ((float)( average_begin_x ) / bm_width * 100 );
  percentage_x += ((float)((long)( bm_width ) - average_end_x ) /
                  bm_width * 100 );
  printf( "percentage X savings = %2.2f\n", percentage_x );
#endif

  /* Relying on the loader to execute a def_bitmap instruction       */
  /* beforeloading the GP instruction list generated by this program.*/

  /* Convert each color in the bitmap into scan lines -              */
  /* one color at a time                                             */
  for( i = index = 0; i < MAX_NUM_COLORS; i++ )
  {
     if ( colors[i].count == 0L )      continue; /* skip non-existant */
                                                 /* colors */
     buff_overflow = FALSE;
     n = extract_scan_lines((long)(index << 1),colors, i, buff,
                                                 &buff_overflow);

     if ( buff_overflow )
     {
          fprintf( stderr, "GP instruction list overflow.\n" );
          exit( 1 );
     }
     /* If the newly extracted scan lines array can't fit into the  */
     /* GP instruction buffer, store instruction built up so far,   */
     /* and start filling the buffer from the begining.             */
     if (( index + n ) > GP_BUFF_SIZE )
     {
        /* Flag the user if a color generates more lines than there */
        /* is space in the instruction buffer. Very unlikely.       */
          if ( index <= 0 )
          {
               fprintf( stderr, "Instruction list overflow.\n" );
               exit( 1 );
          }
          /* store GP instruction built up so far */
          write_buffer_to_file( f2, gp_buff, index );
```

Listing One *(Listing continued, text begins on page 42.)*

```
          /* adjust the addresses in the GP instruction set */
          /* since the GP code is not relocatable.          */
          index = 0;
          buff[ 4 ] = (int)(( 20L ) & 0xffffL ); /* scan line array  */
                                                 /* address */
          buff[ 5 ] = (int)(( 20L ) >> 16 );
          buff[ 8 ] = (int)(( (long)( n << 1 )) & 0xffffL );
                                                 /* link address */
          buff[ 9 ] = (int)(( (long)( n << 1 )) >> 16);
     }
     /* copy elements from temporary buffer into instruction buffer  */
     for ( j = 0; j < n; )
          gp_buff[ index++ ] = buff[ j++ ];
#if DEBUG
     printf( "index = %d\n", index );
#endif
  }
  /* store whatever is left in the very last buffer */
  if ( index > 0 )
     write_buffer_to_file( f2, gp_buff, index );

#if MOVSW_ON
  /* XENIX specific - disable movesw in the 82786 driver */
  if ( regdata.rdata.regval == FALSE )
     ioctl( p6handle, CLEAR_SFLAG, &value );
#endif

  return( 0 );          /* DONE!!! Wasn't that simple?! */
}
/*-------------------------------------------------------------------*/
/* Scan through the bitmap once and fill the 'colors' array with     */
/* some very useful data about each color - how many times it apears */
/* in the bitmap and where in the bitmap it resides (define a window */
/* of existance for each color. Return number of colors that were    */
/* found in the bitmap.                                              */
/*-------------------------------------------------------------------*/
find_all_colors( colors, num_lines )
color_t colors[];   /* array of colors - 256 elements */
int     num_lines;  /* number of lines in the bitmap  */
{
     register int     x;          /* x coordinate on a scan line     */
     register color_t *color_ptr; /* pointer - for speed             */
     register int     n;          /* number of bytes in a scan line  */
     int  line,                   /* present scan line in the bitmap */
          num_colors;             /* number of colors found in the   */
                                  /* bitmap                          */

#if DEBUG_1
     printf("Entered find_all_colors routine. num_lines = %d\n",
                                                      num_lines );
#endif
     /* Initialize the 'colors' array. */
     for( x = 0; x < MAX_NUM_COLORS; )
     {
          color_ptr = &colors[x++];       /* use a pointer for speed */
          color_ptr->count = 0L;
          color_ptr->begin_y = color_ptr->end_y = 0;
          color_ptr->begin_x = color_ptr->end_x = 0;
     }

     /* Scan and analyze the bitmap one line at a time. */
     for ( line = 0; line < num_lines; line++ )
     {
          n = get_scan_line( bm_address, line_buff, line, bm_width );
          for( x = 0; x < n; x++ )
          {
               color_ptr = &( colors[ line_buff[x] ]);

               /* mark the begining scan line for this color */
               if ( color_ptr->count++ == 0L )
               {
                    color_ptr->begin_y = line;
                    color_ptr->begin_x = x;
               }
               /* adjust the ending scan line each time a color is   */
               /* detected                                           */
               color_ptr->end_y = line;

               /* adjust x window for a color if needed */
               if ( x < color_ptr->begin_x )    color_ptr->begin_x = x;
               if ( x > color_ptr->end_x   )    color_ptr->end_x   = x;

          }
     }

     for ( x = num_colors = 0; x < MAX_NUM_COLORS; )
          if ( colors[x++].count > 0L )     num_colors++;

#if DEBUG_1
     printf( "Exited find_all_colors routine.\n" );
#endif
     return( num_colors );
}
/*-------------------------------------------------------------------*/
/*              The heart of compression.                            */
/* Procedure to extract scan lines from a bitmap file (with some     */
/* help from 'colors' array). Assumes that the GP buffer is          */
/* impossible to overrun (left as an exercise to correct). The best  */
/* way to understand this one is to go through it with a particular  */
/* bitmap in mind.                                                   */
/*-------------------------------------------------------------------*/
extract_scan_lines( start_addr, colors, color, buff, overflow )
long  start_addr;   /* starting address of this GP instruction list  */
color_t  colors[];  /* colors description array                      */
int      color,     /* color that is being extracted                 */
     buff[],        /* gp instruction buffer                         */
```

(continued on page 88)

Listing One *(Listing continued, text begins on page 42.)*

```
     overflow;      /* overflow flag, qets set if the instruction     */
                    /* buffer end is reached = ( num_elements -       */
                    /* GP_BUFF_THRESHOLD )                            */
{
     /* Keep x and y coordinates from call to call - needed to        */
     /* calculate dx and dy of the next scan line array.              */
     static     int     x = 0,              /* present x coordinate */
                        y = 0;              /* present y coordinate */
     register     i, count, line;
     int  index, dy;                /* gp instruction buffer index */
     int  n, num_lines,
          num_lines_index, first_time;
     int  link_lower, link_upper;
     BOOLEAN  within_scan_line;

#if DEBUG
     printf( "color = %d\n",color );
#endif

     /* Start at the begining of a buffer and add the GP              */
     /* instructions to define color, scan lines, and link (around    */
     /* the array). Relies on the loader to def_bitmap, texture, and  */
     /* raster operation.                                             */
     index = 0;

     /* def_color instruction */
     buff[ index++ ] = 0x3d00;
     buff[ index++ ] = (((int)color ) | (((int)color ) << 8));
     buff[ index++ ] = 0;

     /* scan_lines instruction */
     buff[ index++ ] = 0xba00;
     buff[ index++ ] = (int)(( start_addr + 20L ) & 0xffffL );
     buff[ index++ ] = (int)(( start_addr + 20L ) >> 16 );
     num_lines_index = index++; /* number of lines in the scan lines */
                                /* array is not yet known.           */

     /* link instruction - jump around the array */
     buff[ index++ ] = 0x0200;
     link_lower = index++;     /* fill in when the number of elements */
     link_upper = index++;     /* is known.                           */

     num_lines = 0;
     first_time = TRUE;

     /* start at the bottom of the window (of this color)             */
     /* and process one line at a time until the top of the window.   */
     dy = line = colors[ color ].begin_y;
     for ( ; line <= colors[ color ].end_y; line++ )
     {
          n = get_scan_line( bm_address, line_buff, line, bm_width );
          count = 0;
          within_scan_line = FALSE;

          /* Process the line one pixel at a time */
          n = colors[ color ].end_x;
          for( i = colors[ color ].begin_x; i <= n; i++ )
          {
               if ( line_buff[i] != color )
               {
                    /* found a pixel that is not of desired color */
                    if ( within_scan_line )
                    {
                         /* reached the end of scan line of desired  */
                         /* color                                    */
                         buff[ index++ ] = --count;  /* length of it */
                         within_scan_line = FALSE;
                         y += dy;
                         count = dy = 0;
                         num_lines++;
                    }
                    continue;      /* to the next pixel */
               }
               else     /* found a pixel of desired color */
               {
                    if ( ! within_scan_line )  /* found the begining */
                    {
                         buff[ index++ ] = i - x; /* dx for scan line*/
                                                  /* instruction */
                         x = i;
                         if ( first_time )
                         {
                              /* first time for this color */
#if DEBUG_2
     printf( "first time, y = %d, dy = %d\n", y, dy );
#endif
                              buff[ index++ ] = dy - y; /* dy for    */
                                                        /* scan line */
                              y = dy;
                              dy = 0;          /* reset dy, now that */
                                               /* we've moved        */
                              first_time = FALSE;
                         }
                         else
                              buff[ index++ ] = dy; /* dy for scan   */
                                                    /* line instr.   */
                         within_scan_line = TRUE;   /* signal the    */
                                                    /* begining edge */
                    }
                    count++;

                    /* Take care of the last pixel == color case */
                    if ( i == n )
                    {
                         buff[ index++ ] = --count;
                         within_scan_line = FALSE;
                         y += dy;
                         count = dy = 0;
                         num_lines++;
                    }
               }
          }
#if DEBUG_1
          printf( "x = %d,\t y = %d\n", x, y );
#endif
          dy++;
     }
     /* Now, the number of lines of this color is known.              */
     /* Therefore, scan line array instruction and link address       */
     /* can be filled.                                                */
     buff[ num_lines_index ] = num_lines;
     buff[ link_lower ] = (int)(( start_addr + (long)( index << 1))
                                                        & 0xffffL );
     buff[ link_upper ] = (int)(( start_addr + (long)( index << 1))
                                                           >> 16);
#if DEBUG_2
     printf( "num_lines = %d,\tx = %d,\t y = %d\n", num_lines, x, y );
#endif
     return( index );
}
/*-------------------------------------------------------------------*/
/* Procedure that writes the GP instruction list to a file.          */
/* An appropriate header is added before the GP list.                */
/*-------------------------------------------------------------------*/
write_buffer_to_file( fd, buff, num_of_elements )
int  fd,                  /* output file descriptor                  */
     buff[],              /* pointer to the buffer                   */
     num_of_elements;     /* number of elements to be written        */
                          /* each element is 16 bits (integer)       */
{
     /* Header - placed before every block (8 bytes) */
     struct     header
     {
          int       type;       /* 0 - GP instructions, 1 - bitmap     */
          long      addr;       /* load address, ffffffff - don't care */
          int       num_bytes; /* number of bytes                      */
     };
     typedef     struct header     header_t;
     header_t     hdr;

     /* Write the header into the file */
     hdr.type = 0;
     hdr.addr = 0L;          /* tell the loader to place instructions */
                             /* address 0 in 82786 memory.            */
     hdr.num_bytes = num_of_elements << 1;

     /* Write the header into the output file */
     if ( write( fd, &hdr, sizeof( hdr )) != sizeof( hdr ))
     {
          fprintf( stderr, "compact: Write error.\n" );
          exit(1);
     }

     /* Write the GP instruction list into the output file */
     if ( write( fd, buff, num_of_elements << 1 ) != ( num_of_elements
                                                        << 1 ))
     {
          fprintf( stderr, "compact: Write error.\n" );
          exit(1);
     }
     return( OK );
}
/*-------------------------------------------------------------------*/
/* Procedure to read any scan line from the bitmap stored in         */
/* graphics memory. Swap bytes to make scanning easier.              */
/*-------------------------------------------------------------------*/
get_scan_line( base_addr, buff_gsl, line, line_width )
long  base_addr;          /* starting address of the bitmap          */
unsigned char  *buff_gsl;   /* scan line buffer                      */
int   line,               /* which line to read                      */
      line_width;         /* how many pixels in a line               */
{
     long     address;

#if DEBUG_1
     printf( "Entered get_scan_line routine. addr = %lx", addr );
     printf( "\tline = %d\tline_width = %d\n", line, line_width );
#endif

     address = base_addr + ((long)( line ) * (long)( line_width ));
     getmem( p6handle, address, buff_gsl, line_width >> 1 );
     swab( buff_gsl, buff_gsl, line_width );

     /* be carefull with swab (note that source and destination are   */
     /* the same) functionality depends on implementation of the      */
     /* swab routine.                                                 */

#if DEBUG_1
     printf("Exited get_scan_line routine.\n");
#endif
     return( line_width );
}
```

End Listing One

Listing Two

```
/*           A simple GP instruction list loader.                   */
/*         Created by Victor J. Duvanenko                           */
/*                                                                  */
/* Loads a GP instruction list from a file into 82786 memory and    */
/* instructs the GP to execute them. If the file contains more      */
/* instructions they are read in. The loader then waits for the GP  */
/* to finish the previous list. Only when the GP is finished does   */
/* the loader place the new list in 82786 memory.                   */
/*                                                                  */
/* Usage:           load file_name                                  */
/*                                                                  */
```

(continued on page 90)

Listing Two *(Listing continued, text begins on page 42.)*

```
/* Input:  Binary file of the followind format                          */
/*         a) Header                                                    */
/*            1) type ( 0 - GP instructions, 1 - bitmap data ).         */
/*            2) 32 bit address (lets the loader know where to place    */
/*               the data in graphics/82786 memory). If equal to        */
/*               0xffffffff then the loader can place data anywhere.    */
/*            3) number of bytes to load.                               */
/*         b) Data                                                      */
/*            1) GP instruction list.                                   */
/*                                                                      */
/* Output:  GP instructions are loaded into 82786 memory.               */
/*                                                                      */
/* The loader provides the following services                           */
/* (may harm some applications)                                         */
/* 1) def_bitmap, def_texture, and raster_op instructions with          */
/*    certain defaults are executed before loading the GP               */
/*    instruction list.                                                 */
/* 2) "stop" instruction is placed at the end of every GP list -        */
/*     'nop' with GCL bit set.                                          */
/* 3) Load time quote in milliseconds.                                  */

#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/timeb.h>
#include "/usr/tls786/h/tls786.h"
#include "/usr/tls786/h/tvl786.h"


#define BUFF_SIZE     32600
#define BOOLEAN         int
#define TRUE              1
#define FALSE             0
#define OK                1
#define INTERVAL          1       /* Sampling period in milliseconds */
#define WAIT           5000/* wait period for the GP or DP to finish */
#define COMM_BUF_BOTTOM  0L

#define DEBUG         FALSE
#define DEBG_1        FALSE

/* GP instruction list buffer */
unsigned     char     buff[ BUFF_SIZE ];

main(argc,argv)
int argc;
char *argv[];
{
     register i;
     int p6handle, fl, n_items, ellapsed_time, value;
     struct timeb time_before, time_after;
     long     addr,
          addr_bm;    /* bitmap base address */

     /* Header - placed before every block (8 bytes) */
     struct     header
     {
          int      type;      /* 0 - GP instructions, 1 - Bitmap     */
          long     addr;      /* load address, ffffffff - don't care */
          int      num_bytes; /* number of bytes                     */
     };
     typedef     struct header     header_t;

     header_t     hdr;

     /* XENIX specific - turns on movsw instruction */
     union
     {
          struct  tvl_cntrl brddata;
          byte    rawdata[ sizeof( struct tvl_cntrl )];
          struct  io_data   rdata;
     }regdata;

   /* Check command line for proper usage - just a little. */
   if (argc == 1)
   {
     fprintf( stderr, "Usage is: %s file_name\n", argv[0] );
     exit(1);
   }

   /* Open the input file for reading only. */
   if (( fl = open( argv[1], O_RDONLY ))  == -1 )
   {
      fprintf( stderr, "%s: Can't open %s.\n", argv[0], argv[1] );
      exit(1);
   }

   /* XENIX specific - enable the 82786 driver. */
   p6handle = open786( MASTER_786 );
   if ( p6handle == NULL )     {
      fprintf( stderr, "%s: Can't access 82786.\n", argv[0] );
      exit(1);
   }

   /* XENIX specific - enable the use of movsw instruction driver. */
   value = 0;
   ioctl( p6handle, TEST_SFLAG, &regdata );
   if ( regdata.rdata.regval == FALSE )
      ioctl( p6handle, SET_SFLAG, &value );

   addr = 0L;
   addr_bm = 0x10000L;
   ftime( &time_before );            /* Get present time stamp */

   /* A bit of overhead to make sure that the bitmap and texture are  */
   /* defined before the GP command list is loaded.                   */
      i = 0;
      buff[ i++ ] = 0x00;          /* Def_bitmap */
      buff[ i++ ] = 0x1a;
      buff[ i++ ] = 0x00;
      buff[ i++ ] = 0x00;
      buff[ i++ ] = 0x01;
      buff[ i++ ] = 0x00;
      buff[ i++ ] = 0x7f;          /* 640 (for now) */
      buff[ i++ ] = 0x02;
      buff[ i++ ] = 0xdf;          /* by 480 (for now) */
      buff[ i++ ] = 0x01;
      buff[ i++ ] = 0x08;          /* 8bpp (for now) */
      buff[ i++ ] = 0x00;

      buff[ i++ ] = 0x00;          /* Def_logical_op */
      buff[ i++ ] = 0x41;
      buff[ i++ ] = 0xff;
      buff[ i++ ] = 0xff;
      buff[ i++ ] = 0x05;
      buff[ i++ ] = 0x00;

      buff[ i++ ] = 0x00;          /* Def_texture */
      buff[ i++ ] = 0x06;
      buff[ i++ ] = 0xff;
      buff[ i++ ] = 0xff;

      buff[ i++ ] = 0x01;          /* stop */
      buff[ i++ ] = 0x03;

      /* Wait for a previous GP command list to finish */
      if ( waitgp( p6handle, INTERVAL, WAIT ) < 0 )     {
           printf("GP is hung!!!\n");
           exit(1);
      }
        /* Place it in 786 graphics memory */
        putmem( p6handle, addr, buff, i >> 1 );

      /* Direct the GP to execute the command */
      putreg( p6handle, GRP_GR1, (int)( addr & 0xffff ));
      putreg( p6handle, GRP_GR2, (int)( addr >> 16 ));
      putreg( p6handle, GRP_GR0, 0x200 );

   /* Now, for the GP list from an input file. */
   /* Read the header and then the data.       */
   while (( n_items = read( fl, &hdr, sizeof( hdr ))) > 0 )
   {
      i = 0;
      if ( n_items != sizeof( hdr ))
      {
           printf( stderr, "%s: Read error.\n", argv[0] );
           exit(1);
      }
     /* does it matter where the GP list is placed? */
      if ( hdr.addr != 0xffffffffL )      addr = hdr.addr;

      /* GP instruction list */
      if ( hdr.type == 0 )
      {
           if (( n_items = read( fl, buff, hdr.num_bytes )) ==
                                                    hdr.num_bytes )
           {
                /* Add a "stop" command to the GP instruction list */
                i += n_items;
                buff[ i++ ] = 0x01;
                buff[ i++ ] = 0x03;

                /* Wait for the GP to finish any previous instruction */
                if ( waitgp( p6handle, INTERVAL, WAIT ) < 0 )
                {
                     fprintf( stderr, "GP is hung!!!\n" );
                     exit( 1 );
                }

                  /* Place it in 786 graphics memory */
                  putmem( p6handle, addr, buff, i >> 1 );
           }
           else
           {
                printf( stderr, "%s: Read error.\n", argv[0] );
                exit(1);
           }
           /* Direct the GP to execute the command */
           putreg( p6handle, GRP_GR1, (int)( addr & 0xffff ));
           putreg( p6handle, GRP_GR2, (int)( addr >> 16 ));
           putreg( p6handle, GRP_GR0, 0x200 );
      }
      /* Is it bitmaps - then place the data at that address */
      if ( hdr.type == 1 )
      {
           if (( n_items = read( fl, &buff[i], hdr.num_bytes )) ==
                                                    hdr.num_bytes )
           {
              /* Place it in 786 graphics memory */
              putmem( p6handle, addr_bm + hdr.addr, buff, n_items >> 1 );
           }
      }
   }

   /* Get the time stamp after the loading is done. */
   ftime( &time_after );
   ellapsed_time = (int)( time_after.time - time_before.time ) * 1000;
   ellapsed_time += ( time_after.millitm - time_before.millitm );
   printf( "%dms\n", ellapsed_time );

   /* XENIX specific - disable movesw in the 786 driver */
   if ( regdata.rdata.regval == FALSE )
      ioctl( p6handle, CLEAR_SFLAG, &value );

   return(0);
}
```

End Listings

Listing One *(Text begins on page 50.)*

```
/* Dynamic structure include file */

#ifndef _V_STRUCT_
#define _V_STRUCT_

#include <stdio.h>

char *Var_types[] = { "int", "float", "double", ""};
enum V_TYPES { UNKNOWN=-1, INT, FLOAT, DOUBLE };

#define MAX_VAR       256
#define MAX_NAME       16
#define MAX_DATA     1024

typedef struct {
  char name[MAX_NAME];
  enum V_TYPES type;
} ELEMENT;

typedef struct {
  ELEMENT element[MAX_VAR];
  int count;
  char data[MAX_DATA];
  FILE *source;
} D_STRUCT;

#endif
```

End Listing One

Listing Two

```
#include "d_struct.h"

/* Returns the number of definitions, -1 if */
/* any error occurs */
int load_v_defs(v_file, d_struct)
char v_file[];
D_STRUCT *d_struct;
{
  int cnt;
  char type_text[12];
  char file_name[16];
  FILE *fptr;

/* Open variable definition of variable  */
/* names/record structure */
  strcpy(file_name, v_file);
  strcat(file_name, ".DEF");
  fptr = fopen(file_name, "r");
  if(fptr == NULL) return(-1);

/* Initialize */
  d_struct->count = 0;

/* load in definitions */

  cnt = d_struct->count;
  while(fscanf(fptr, "%s %s", type_text,
    d_struct->element[cnt].name) != EOF)
  {
    d_struct->element[cnt].type = v_type(type_text);
    cnt++;
    if(cnt > MAX_VAR) return(-1);
  }
  d_struct->count = cnt;
  fclose(fptr);

/* Now open data file - leave open for reading */
/* of data */
  strcpy(file_name, v_file);
  strcat(file_name, ".DAT");
  d_struct->source = fopen(file_name, "r");
  if(d_struct->source == NULL) return(-1);
  return(d_struct->count);
}

/* Invalid type returns -1, otherwise index of */
/* type in 'Var_types' */
int v_type(type_text)
char type_text[];
{
  int i = 0;

  while(strlen(Var_types[i]) != 0)
  {
    if(strcmp(type_text, Var_types[i]) == 0) return(i);
    i++;
  }
  return(-1);
}

/* Brings a record into the data workspace */
char *read_data(d_struct)
D_STRUCT *d_struct;
{
  return(fgets(d_struct->data, MAX_DATA, d_struct->source));
}

/* extracts a data value from the record */
/* which corresponds to a dynamic name   */
double get_v_value(v_name, d_struct)
char v_name[];
D_STRUCT *d_struct;
{
  int i;
  double dval;
  char tmp_fmt[256], fmt[256];

  strcpy(fmt, "");
  for(i=0; i< d_struct->count; i++)
  {
    if(strcmp(d_struct->element[i].name, v_name) == 0)
    {
      strcpy(tmp_fmt, fmt);
      strcat(tmp_fmt, "%lf");
      sscanf(d_struct->data, tmp_fmt, &dval);
      return(dval);
    }
    strcat(fmt, "%*lf ");
  }
  return(0l);
}

/* Terminates the all reads - clean up */
int end_read(d_struct)
D_STRUCT *d_struct;
{
  close(d_struct->source);
}


/* Prints a variable - in a format according to its type */
int print_v_value(v_name, d_struct)
char v_name[];
D_STRUCT *d_struct;
{
  int i;

  for(i=0; i<d_struct->count; i++)
  {
    if(strcmp(v_name, d_struct->element[i].name) == 0)
    {
      switch(d_struct->element[i].type)
      {
      case INT:
        printf("%16d ", (int)get_v_value(v_name, d_struct));
        break;
      case FLOAT:
        printf("%16.6f ", (float)get_v_value(v_name, d_struct));
        break;
      case DOUBLE:
        printf("%16.6lf ", (double)get_v_value(v_name, d_struct));
        break;
      default:
        printf("%16s ", "Unknown type");
        break;
      }
      return(1);
    }
  }
  printf("%16s ", "Unknown Variable");
  return(0);
}
```

End Listing Two

Listing Three

```
/* Example program - reads data and prints it on the screen */
/* Command line arguments are:                              */
/*      example <filename> <colname> [<colname> ...]        */

#include "vman_1.c"

main(argc, argv)
int argc;
char *argv[];
{
  int i;
  double dval;
  D_STRUCT d_struct;

  if(argc < 3) {
    printf("Proper usage: example <filename> <colname>  [<colname> ...]
                                                                \n");
    exit(-1);
  }

  load_v_defs(argv[1], &d_struct);
  for(i=2; i<argc; i++) printf("  %12s   ", argv[i]);
  printf("\n");
  while(read_data(&d_struct))
  {
    for(i=2; i<argc; i++)
    {
      print_v_value(argv[i], &d_struct);
    }
    printf("\n");
  }
  end_read(&d_struct);
}
```

End Listings

Listing One *(Text begins on page 56.)*

```
/*
  ================== MCB.C ================================
    This program chains through all the DOS memory control
    blocks and computes and prints out information related to
    each one.
    R.J. Moore (C) 06 June 1988 Version 1.2. May be used freely
    for noncommercial purposes only.
    Compiled under Turbo C, Version 1.5, using the small memory
    model, which requires the explicit declaration of some huge and
    far pointers.  Tested on IBM PC, IBM XT, IBM AT, TP/286 AT
    clone, under PC-DOS 2.1, 3.1, 3.2, and 3.3.  Also run under
    MS-DOS 2.11 on a DEC Rainbow.  PC-DOS 3.3 required some
    adjustments, discussed in later comments.
    ========================================================
*/

#include <stdio.h>
#include <dos.h>

/*-------Global declarations-------------------------------------*/
struct MCB              /*template for a one paragraph MS-DOS MCB    */
{
     char chain;        /* 'Z' for last MCB, 'M' for all others      */
     unsigned  pid;     /* PSP segment for process owning the MCB    */
     unsigned  psize;   /* Paragraphs of memory in the MB following  */
     char unused [11]; /* Last 11 bytes of MCB (currently unused)    */
};

typedef struct MCB huge *PTRMCB;     /*PTRMCB is a type declared
                                       to be a huge pointer to MCB   */

/*-------Function prototypes-------------------------------------*/
void main (void);
void far *ffmcb (void);    /* Returns far pointer to first MCB.      */
void prn_header (void);    /* Prints output table header.            */
void prn_mcb (PTRMCB pm); /* Prints out MCB and related information.*/
                           /* Prints out owner name string for pid.  */
void prn_pid_own (unsigned pid,unsigned parent);

/*------main()-------------------------------------------------------*/
/* Executive to control finding a pointer to each MCB and directing
   the printing out of information for each until the end of the
   MCB chain is reached.                                             */

void main()
{
  PTRMCB ptrmcb;              /* ptrmcb is a huge pointer to an MCB  */

    /* Get pointer to first MCB. Note that ffmcb() returns a far
       pointer, which is then cast to a huge pointer. A far pointer
       is good enough to find the first MCB since a far pointer can
       start at any memory location.  However, the use of this pointer
       in ptrmcb must be huge because MCBs range over more than 64K,
       which is all that a far pointer can handle since the segment
       portion of a far pointer never changes. I, of course, found
       this out the hard way.  Such special declarations can be
       avoided if this program is compiled under the huge memory
       model, but I think the method I used is more instructive.     */

  ptrmcb = (PTRMCB) ffmcb(); /* Get far pointer to first MCB and
                                cast to huge pointer via (PTRMCB).  */
  prn_header ();             /* Print out table header to stdout.   */
  prn_mcb(ptrmcb);           /* Print out information for first MCB.*/

    /* Print out MCB information for each of the remaining MCBs     */
  do
  {
    ptrmcb += ptrmcb->psize + 1;   /* Get pointer to next MCB        */

    /* Each unit increment of ptrmcb corresponds to one paragraph;
       adding ptrmcb->psize thus increments through entire allocated
       memory block following the MCB. Since this doesn't include
       space occupied by the MCB itself, must increment through one
       more paragraph (+ 1) to point to the next MCB.               */

    prn_mcb(ptrmcb);               /* Print out information for MCB */
  } while (ptrmcb->chain == 'M');  /* as long as not at end of chain*/
                                   /*Print out final decoration.    */
  printf ("================================================");
  puts   ("==========");
}

/*------ffmcb()------------------------------------------------------*/
/* Returns a far pointer to the first MCB in memory.  Explict
   declaration of far needed since small model was used to compile,
   as noted in a comment in main.                                    */

void far *ffmcb(void)
{
  union REGS regs;         /* REGS and SREGS defined in dos.h.       */
  struct SREGS sregs;
  unsigned far *segmptr;  /*  Far pointer to segment address of MCB.*/
  regs.h.ah=0x52;         /*  Undocumented MS-DOS function 52H.     */
  intdosx(&regs, &regs, &sregs);  /* ES:BX-2 points to segment
                                     address of first MCB on return
                                     and is copied to segmptr below.*/

  segmptr=MK_FP(sregs.es,regs.x.bx-2);
  return MK_FP(*segmptr,0);       /* Return pointer to MCB itself.  */
}                                 /* Segment pointed to by *segmptr.*/
                                  /* Offset is zero (on paragraph). */

/*----------prn_header()-------------------------------------------*/
/* Prints out header for the output variables describing the
   information for each MCB, which will be subsequently printed
   out by the function prn_mcb().                                    */
*/
void prn_header (void)
{
  printf ("================================================");
  puts   ("==========");
  puts   ("MCB MCB  ID PID      MB PAR- ENV  OWNER");
  puts   ("NO. SEG            SIZE ENT  BLK?");
  printf ("================================================");
  puts   ("==========");

/*    MCB NO. = ordinal number of MCB being processed (1,2,...).
      MCB SEG = segment address (hex) of memory control block.
      ID      = chain id, 'Z' if last MCB, 'M' otherwise.
      PID     = process id, the PSP segment address (hex) of owner of
                the MCB. (PSP always starts on paragraph boundary.)
      MB SIZE = size of the allocated memory block controlled by
                the MCB (the MB immediately follows its associated
                MCB at the next paragraph in memory (decimal bytes).
      PARENT  = segment address (hex) of parent process's PID.
      ENV BLK?= 'Y' if the MCB controls an environment block,
                'N' otherwise.
      OWNER   = string that prints out program associated with
                the PID.
*/
}

/*------prn_mcb()----------------------------------------------------*/
/* Prints out the information associated with the MCB pointer passed
   to it in its argument list
*/

void prn_mcb (PTRMCB pm)
{
  static cnt = 0;           /* Count of number of times parent has
                               been equal to the pid.               */
  static mcbnum = 1;        /* Ordinal # of MCB being printed out.  */
  unsigned parid;           /* Parent id (segment address of parent
                               process).                            */
  unsigned mcbseg;          /* Segment address of MCB (offset is
                               always zero since paragraph aligned).*/
  char envf;                /*Set to 'Y'/'N' if MB is/is not an
                              environment block.                    */
  unsigned envseg;          /*Segment address of pid's environment
                              block.                                */

   /* Get parent id located at pid:16H                             */
  parid = * (unsigned far *) MK_FP (pm->pid,0x16);

  mcbseg = FP_SEG (pm);     /* Segment address of the MCB           */

  envseg = * (unsigned far *) MK_FP(pm->pid,0x2C);  /* segment     */
                            /* Address of pid's environment        */
                            /* located at pid:2CH.                 */

    /* If the MCB segment value plus one equals the environment
       segment address, then the MCB controls the environment
       block (set envf = 'Y'); otherwise set envf = 'N'             */

  envf = mcbseg+1 ==  envseg ? 'Y' : 'N';

    /* Count the number of times parent and pid have been equal
       (when this is true, memory blocks are owned by COMMAND.COM)  */

  if (parid == pm->pid) cnt++;

    /* The above determination of whether an MB is an environment
       block isn't complete for DOS versions 2.0 thru 3.2.  The
       above logic will not identify the master environment block
       owned by the master copy of COMMAND.COM since the value at
       pid:2CH contains zero, not the segment address of the master
       environment.  The logic below uses the fact that the master
       environment follows the master COMMAND.COM in memory. (The
       environment copies for other programs are in memory BEFORE
       the pid they are associated with.) Starting with DOS 3.3
       pid:2CH always points to the environment, even for the
       master COMMAND.COM, so the following is not needed (but it
       doesn't do any harm).                                        */

  if (!envseg && cnt == 2) envf = 'Y';

    /* Print out MCB information except for owner name in the
       following call to printf().                                  */

  printf("%2.2u%06.4X%2.1c%06.4X%7lu%5.4X %-5.1c",
   mcbnum++,mcbseg,pm->chain,pm->pid,(long) pm->psize*16,parid,envf);

/* Call prn_pid_own() to find and print out owner string           */

  prn_pid_own(pm->pid,parid);
}

/*------prn_pid_own()------------------------------------------------*/

/* Prints out owner name string associated with the pid, which is an
   input parameter.  Also needs the parent address as an input to
   identify cases where COMMAND.COM is the owner (true when
   pid=parent). This function also uses the fact that the following
   pid values are special:

   pid = 0 means that MCB is a free block of memory
   pid = 8 means that the MCB is owned by IBMDOS.COM/MSDOS.SYS
   pid = parent means that the MCB is owned by COMMAND.COM (the only
         program that is its own parent.)
```

(continued on page 96)

MAPPING DOS

Listing One *(Listing continued, text begins on page 56.)*

```
   In these cases I assign appropriate owner string names instead of
   getting them from the environment since they are not available
   there.  Owner names consisting of a string with the drive, path,
   and file name of the program that owns the memory are only
   available in DOS 3.x. Note that DOS 3.3 does not provide an owner
   string for the master copy of COMMAND.COM for some reason.  This
   is of no consequence in the method used here.
*/

void prn_pid_own (unsigned pid,unsigned parent)
{
  unsigned far *envsegptr;  /* Pointer to seg address of environment*/
  char far *envptr;         /* Pointer to pid's environment         */
  unsigned far *envsizeptr; /* Pointer to envsize word below        */
  unsigned envsize;         /* Size of pid's environment            */

  /* Ordinal # of copy of COMMAND.COM in memory (ccnum=1 for master
     copy, 2 for first secondary copy (if any), etc.               */

  static unsigned char ccnum = 0;

  /* Pid value saved from previous call to this function.
  Initialized to an impossible value (no PSP could start at FFFF:0) */

  static prev_pid = 0xFFFF;

  switch (pid)
  {
       /* Assign owner names for two special cases                 */
     case 0 : puts ("FREE MEMORY CONTROL BLOCK");return;
     case 8 : puts ("IBMDOS.COM/MSDOS.SYS");return;
  }

  /* pid:2CH contains ptr to segment address of pid's environment   */
  envsegptr = (unsigned far *) MK_FP (pid,0x2C);

  /* Get pointer to the environment block itself                    */
  envptr = (char far *) MK_FP (*envsegptr,0);

  /* Define a pointer that contains the size of the environment
     block. Must point back one paragraph (where the environment's
     MCB resides) plus three bytes forward (where the MCB block
     size field is).                                                */
  envsizeptr = (unsigned  far *) MK_FP(*envsegptr-1,0x3);

  /* Get the size of the environment using the above pointer in
     units of bytes (1 paragraph = 16 decimal bytes).               */
  envsize = *envsizeptr*16;

    /* If next stmt is satisfied, owner is a copy of COMMAND.COM    */

  if (pid == parent)
  {
    /* If previous pid is different from current pid, have found a
       new secondary copy of COMMAND - ccnum keeps track records of
       the copy number.                                             */

    if (prev_pid != pid) ccnum++;
    printf ("COMMAND.COM COPY #%-2u\n",ccnum);

    prev_pid = pid;         /* Save current pid - will be previous  */
    return;                 /*  in the next call to this function   */
  }

  /* Loop at most until the end of the environment                  */

  while (envsize)
  {
      /* Decrement counter (envsize) indicating # of bytes left in
         environment and advance pointer thru environment block until
         either end of environment or a NULL is located
      */
    while (--envsize && *envptr++);

      /* The next stmt will be true if another NULL immediately
         follows the first one located and a word count of 0001 then
         follows that.                                              */

    if (!*envptr && *(unsigned far *) (envptr+1) == 0x1)
    {
       envptr +=3;            /* Correct pattern found (00 00 01 00) */
       break;                 /* so point envptr to owner string     */
    }
  }

  if (envsize)
  {
     /* If an owner string was found before the end of the
        environment then print out the owner name.  Note that
        can't use puts() or printf() to print out the results

        since I used the small memory model.
      */
    while(*envptr) putchar(*envptr++);
    putchar('\n');
  }
  else
     /* If reached the end of the environment without finding
        an owner string (should only occur for DOS 2.x)             */
    puts ("UNKNOWN OWNER");
}
```

End Listing

SMALLTALK /PROLOG

Listing One *(Text begins on page 68.)*

```
/*  blocks transfer program */


      /* generic */
go( Start, Goal) :- path( Start, Goal, [Start]), !.


path( Goal, Goal, Hist) :-   printpath( Hist).
path( State, Goal, Hist) :-  move( State, Interm),
                             not( member( Interm, Hist) ),
                             path( Interm, Goal, [Interm|Hist]).


member( X, [X|_]).
member( X, [_|T]) :- member( X, T).


printpath( []).
printpath( [H|T] ) :- printpath( T), write( H), nl.



      /* problem specific */
blocks1 :- go( [on(a,b), on(b,t), on(c,t)], [on(a,b), on(b,c), on(c,t)] ).
blocks2 :- go( [on(a,t), on(b,t), on(c,a)], [on(a,b), on(b,c), on(c,t)] ).
blocks3 :- go( [on(a,b), on(b,t), on(c,t),on(d,a)], [on(a,d), on(b,c),
               on(c,t), on(d,b)] ).


move( State1, State2) :- member( on( X,Y), State1),
                         clear( X, State1),
                         not( table( Y)),
                         subst( on( X, Y), State1, on( X, t), State2).
move( State1, State2) :- member( on( X,Y), State1),
                         clear( X, State1),
                         member( on( Z, _), State1),  X \= Z,
                         clear( Z, State1),
                         subst( on( X, Y), State1, on( X, Z), State2).


clear( X, State) :- not( member( on( _, X), State)).


subst( _, [], _, []).
subst( X, [X|L], A, [A|M]) :- !, subst( X, L, A, M).
subst( X, [Y|L], A, [Y|M]) :- subst( X, L, A, M).


table( t).
```

End Listing One

Listing Two

```
Prolog variableSubclass: #Blocks
  instanceVariableNames:
    'position animator number between replyStream selectedChoice '
  classVariableNames: ''
  poolDictionaries: ''


Blocks class methods



Blocks methods

add: block1 onBl:block2
       "add one block to the top of the other"
    | ordColl index col size|
 index := 1.
 col := 0.
 [index <= number
    and: [col = 0] ]
       whileTrue: [
            ( (position at: index) includes: block2 )
                 ifTrue: [ col := index].
            index := index + 1].
 ordColl := position at: col.
 ordColl add: block1.
 size := ordColl size.
 position at: col
          put: ordColl.
 animator tell: block1
           place: ( (between * col) - (between * (2/3) ) )  @
         ( (RectPict extent y - 5) - ( 60 * size * Aspect) truncated )


add: block onTbl: col
       "(re)initialize column: add the first block to it"
    | ordColl |
 ordColl := position at: col.
 ordColl add: block.
 position at: col
          put: ordColl.
 animator tell: block
           place: ( (between * col) - (between * (2/3) ) )  @
       ( (RectPict extent y - 5) - ( 60 *  Aspect) truncated)


assign: size
```

```
        "assign a variable number"
    number := size

choice: aSymbol
        "Private - Change to the selected choice type."
    selectedChoice := aSymbol.      " #blocks1, #blocks2 or #blocks3"
    self
        changed: #input;
        changed: #reply;
        changed: #graph:


choices
        "Private - Answer an Array of choices"
    ^#( blocks1 blocks2 blocks3 )


doBlocks
     "Actual call to Prolog"
    CursorManager execute change.
    selectedChoice == #blocks1
       ifTrue: [self :? blocks1() ].
    selectedChoice == #blocks2
       ifTrue: [self :? blocks2() ].
    selectedChoice == #blocks3
       ifTrue:  [self :? blocks3() ].
    CursorManager normal change


doBlocksMenu
        "Menu do\help"
   ^Menu
       labels: 'do\help' withCrs
       lines: #()
       selectors: #(doBlocks help)


finish
    Menu message: 'The solution is found'


graph: aRect
   " initialize graph pane, assign global variables"
    | aForm |
  aForm := Form
     width: aRect width
     height: aRect height.
  aForm displayAt: aRect origin.          "background"
  RectPict := aRect.                      "global vars"
  White := ( Form
       width: 60 height: ( 60 * Aspect) truncated ).
  ^aForm

help
     "Provide help message"
```

```
    selectedChoice == #blocks1
       ifTrue: [replyStream nextPutAll:
'EXPLANATION: This is an animation of the 3 blocks problem'; cr ].
    selectedChoice == #blocks2
       ifTrue: [replyStream nextPutAll:
'EXPLANATION: This is an animation of the 3 blocks problem'; cr ].
    selectedChoice == #blocks3
     ifTrue: [replyStream nextPutAll:
'EXPLANATION: This is an animation of the 4 blocks problem'; cr ]


initAnimation
   " initialize Animation"
    | blockImages |
  blockImages :=
        Array with: White.
  animator := Animation new
        initialize: RectPict.
  animator add: blockImages
        name: 'Black'
        color: #black.
  animator add: blockImages
        name: 'LightGray'
        color: #lightGray.
   animator add: blockImages
        name: 'Gray'
        color:#gray.
   selectedChoice ==  #blocks3
     ifTrue: [ animator add: blockImages
                   name: 'DarkGray'
                   color: #darkGray].
      " set speed, shift, background"
   animator
     setBackground;
     speed: 8;
     shiftRate: 10


initialize1
          "initialize Blocks"
    | pen|
     " draw bottom"
 pen := Pen new.
 pen defaultNib: 3 @ 2.
 pen
     place: (RectPict origin x + 5) @ (RectPict corner y - 5);
     goto: (RectPict corner x - 5) @ (RectPict corner y - 5).
     " assign between variable"
 between := RectPict width // number.
     "initialize animation and position"
 self
        initAnimation;
        initPosition

initPosition
        "Set the receiver's initial position"
```

```
    position := Array new: number.
    1 to: number do: [:index|
                      position at: index
                              put: OrderedCollection new]


   input
          "Private - Answer an input text for
           the selected choice (blocks1, blocks2 or blocks3)."
     | text1 text2 text3 text|
     text1 := 'FROM:  A on B, B on Table, C on Table
   TO   :  A on B, B on C, C on Table
          ( COLORS: A- Black, B- LightGray, C- Gray)'.
     text2 := 'FROM:  A on Table, B on Table, C on A
   TO   :  A on B, B on C, C on Table
          ( COLORS: A- Black, B- LightGray, C- Gray)'.
     text3 := 'FROM:  A on B, B on Table, C on Table, D on A
   TO   :  A on D, B on C, C on Table, D on B
    ( COLORS: A- Black, B- LightGray, C- Gray, D- DarkGray)'.
     selectedChoice == #blocks1
        ifTrue: [text := text1].
     selectedChoice == #blocks2
        ifTrue: [text := text2].
     selectedChoice == #blocks3
        ifTrue: [text := text3].
     ^text
   open On
          "Create a window on Blocks.
           Define the type, behavior and relative
           size of each pane and schedule the window."
       | topPane replyPane|
       topPane := TopPane new label: 'B L O C K S'.
       topPane addSubpane:
           (ListPane new
              model: self;
              name: #choices;
              change: #choice:;
              selection: 1;
              framingRatio: ( 0 @ 0 extent: 1/4 @ (1/6) ) ).
       selectedChoice := #blocks1.
       topPane addSubpane:
           (TextPane new
              model: self;
              name: #input;
              menu: #doBlocksMenu;
              framingRatio: ( 1/4 @ 0 extent:  3/4 @ (1/6) ) ).
       topPane addSubpane:
           (replyPane := TextPane new
              model: self;
              name: #reply;
              framingRatio: ( 0 @ (1/6) extent: 1 @ (1/6) ) ).
       topPane addSubpane:
           (GraphPane new
              model: self;
              name: #graph:;
              framingRatio: ( 0 @ (1/3) extent: 1 @ (2/3) ) ).
       topPane reframe:
           (Display boundingBox insetBy: 10@10).
       replyStream := replyPane dispatcher.
       topPane dispatcher openWindow scheduleWindow


   remove: block
          "remove block (from the data structure only)"
       | ordColl index col|
     index := 1.
     col := 0.
     [index <= number
        and: [col = 0] ]
          whileTrue: [
              ( (position at: index) includes: block )
                  ifTrue: [ col := index].
              index := index + 1].
     ordColl := position at: col.
     ordColl removeLast.
     position at: col
            put: ordColl


   reply
       " Initiate reply pane with an empty String."
     ^ String new
```

End Listing Two

(continued on page 100)

Listing Three *(Text begins on page 68.)*

```
Blocks variableSubclass: #BlocksPro
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''


BlocksPro class methods


BlocksPro methods

"add on the Table"
addDraw( x) :-
    name( x, name1), col( x, col1),
    is( _, self add: name1 value onTbl: col1 value).


"add on the Block"
addDraw( x, z) :-
    name(x, name1), name( z, name2),
    is( _, self add: name1 value onBl: name2 value).


"arrange0 - the first part of arrange"
arrange0( len, _, _, accum, accum) :-
            length( accum, accumLen),
            eq( len, accumLen).
arrange0( len, master, prev, accum, ordered) :-
            nextStep( master, prev, [], nextPrev),
            append( nextPrev, accum, nextAccum),
            arrange0( len, master, nextPrev, nextAccum, ordered).


"arrange1 - the second step of arrange"
arrange1( master, [#t], arrMaster, arrMaster).
arrange1( master, [h|t], interm, arrMaster) :-
            member( on(h,x), master),
            arrange1( master, t, [on(h,x)| interm], arrMaster).


"arrange ,i.e. [on(a,b),on(b,t),on(c,t),on(d,a)] -->
    --> [d,a,b,c,t] --> [on(c,t), on(b,t), on(a,b), on(d,a)] "
arrange( master, arrMaster) :-
             length( master, len0),
             is( len, len0 value + 1),
             arrange0( len, master, [#t], [#t], ordered),
             arrange1( master, ordered, [], arrMaster),
             !.


"blocks1"
blocks1():- go( [on(#a,#b), on(#b,#t), on(#c,#t)],  [on(#a,#b),
               on(#b,#c), on(#c,#t)] ).


"blocks2"
blocks2():- go( [on(#a,#t), on(#b,#t), on(#c,#a)],  [on(#a,#b),
               on(#b,#c), on(#c,#t)] ).


"blocks3"
blocks3():- go( [on(#a,#b), on(#b,#t), on(#c,#t), on(#d,#a)],
               [on(#a,#d), on(#b,#c), on(#c,#t), on(#d,#b)] ).


"clear"
clear( x, state) :- not( member( on( _, x), state) ).


"column"
col( #a, 1).
col( #b, 2).
col( #c, 3).
col( #d, 4).


"go"
go( start, goal) :- init( start),
                    path( start, goal, [start]),
                    is( _, self finish),
                    !.


"initialize"
init( start) :- length( start, startSize),
                is( _, self assign: startSize value),
                is( _, self initializel ),
                arrange( start, arrStart),
                initDraw( arrStart),
                is( _, Menu message: 'Press button to start'),
                !.


"initial drawing"
initDraw( []).
initDraw( [on(x,#t)|tail]) :- addDraw( x),
                              initDraw( tail).
initDraw( [on(x,z)|tail]) :-  addDraw( x,z),
                              initDraw( tail).
"move"
move(state1,state2,x,y,#t) :- member( on( x, y), state1),
                      clear( x, state1),
                      not( table( y) ),
                      subst( on( x, y), state1, on( x, #t), state2).
move(state1,state2,x,y,z) :- member( on( x, y), state1),
```

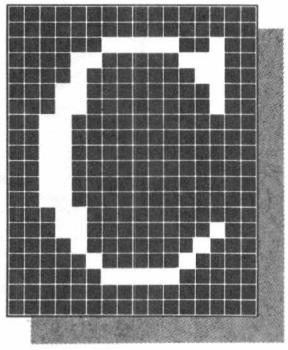
SMALLTALK/PROLOG

Listing Three *(Listing continued, text begins on page 68.)*

```
                      clear( x, state1),
                      member( on( z, _), state1),  ne( x, z),
                      clear( z, state1),
                      subst( on( x, y), state1, on( x, z), state2).


      "draw Block --> Table"
moveDraw( x, _, #t) :-
    name( x, name1), col( x, col1),
    is( _, self remove: name1 value),
    is( _, self add: name1 value onTbl: col1 value).

      "draw Table --> Block; Block --> Block"
moveDraw( x, _, z) :-
    ne( z, #t),
    name(x, name1), name( z, name2),
    is( _, self remove: name1 value),
    is( _, self add: name1 value onBl: name2 value).


"draw In and Out"
moveDrawInOut( block, placeFrom, placeTo) :-
                        moveDraw( block, placeFrom, placeTo).
moveDrawInOut( block, placeFrom, placeTo) :-
                        moveDraw( block, placeTo, placeFrom),
                        fail().                    " reverse back"


"name"
name( #a, 'Black').
name( #b, 'LightGray').
name( #c, 'Gray').
name( #d, 'DarkGray').


"nextStep - called from arrange0"
nextStep( _, [], nextPrev, nextPrev).
nextStep( master, [h|t], current, nextPrev) :-
      findall( x, member( on( x,h), master), interm),
      append( interm, current, current1),
      nextStep( master, t, current1, nextPrev).


"path"
path( goal, goal, hist) :- is( _, self changed: #reply),
                           printpath( hist).
path( state, goal, hist) :- move(state, interm, block, placeFrom,
                            placeTo), not( member( interm, hist) ),
                            moveDrawInOut( block, placeFrom, placeTo),
                            path( interm, goal, [interm| hist] ).

"printpath1"
printpath1( [h]) :- is( _, replyStream nextPutAll:
                              (h value printString) ).
printpath1( [h| t] ) :- is( _, replyStream nextPutAll:
                              (  (h value printString), ', ' ) ),
                        printpath1( t).


"printpath - print list in the reverse order"
printpath( []).
printpath( [h| t] ) :- printpath( t),
                       is( _, replyStream nextPutAll: '['),
                       printpath1( h),
                       is( _, replyStream nextPutAll: ']'),
                       is( _, replyStream cr).


"substitute"
subst( _, [], _, []).
subst( x, [x| l], a, [a| m]) :- !,
                                subst( x, l, a, m).
subst( x, [y| l], a, [y| m]) :- subst( x, l, a, m).


"table"
 table( #t).
```

End Listings

A Window Text Editor



by Al Stevens

This month adds a text editor to the PC C toolset that we are building. The past two "C Programming" columns contained a window library, data entry windows, and a menu manager. Future columns will add help windows, communications functions, and file transfer protocols to the package. Eventually it will all be gathered together into an integrated utility program. But before we get to this month's text editor, I want to deal with a reader concern which has been steadily gaining in volume.

Several readers have expressed their dismay that this project is locked in Turbo C, therefore locking them out. I recognize this as a valid concern and will address this in the January 1989 column.

The correction will take the form of a library of functions and macros that translate the Turbo C specific code into code acceptable to Microsoft's C. In the January column I will describe the library so that users of compilers can develop similar libraries. Now back to business.

To design a text editor, we need to examine our requirements, first to see why we need one and then to see what it should do. Probably nothing in computing is as personal as your editor, whether you use it for word processing or for entering source code. Because you and I are the users of this package, and because each of us already has a preferred editor and word processor, we should question our motives for adding yet another one to the selection. Then, assuming we find justification for another editor, its design must cater to the particular requirements of each of us. Not an easy objective, but a noble one, nonetheless.

The program that results from all this will be a communications utility package for access to an on-line service. On-line services involve the exchange of text —mail, forum conversations, and messages to SYSOPs —between the users. Therefore, an access program needs the entry of free-form text, in other words, a text editor. We could specify that a user can use his or her own editor, and we could provide a gateway to it in our code. Perhaps that feature is an additional requirement for our system. Even so, no doubt there will be other places in the program where text entry is required, places where it would be awkward to exit to a text editor and attach the resulting text data entry to the data structures being built. Real-time conference conversation processing might be one such application. It seems, therefore, that we need a window-oriented text editor, one that can be called from a program to collect text through a window into a buffer.

Because editors usually operate from a command language, and because most users have personal preferences for editing commands, our program must also allow the user to modify the command set.

Listing One on page 119 and Listing Two on page 119 are editor.h and editor.c. These constitute the basic window text editor. Next month we will add a pop-up menu shell, text searching functions, and some file management features to create, edit, save, and merge text files. The expanded editor program will constitute a tiny word processor, one that can be integrated into a C application exactly the way we intend to use it for mail and forum messages. As with all our tools, the editor is designed to be reusable for other projects that you or I might undertake.

Before I describe the code, let's discuss the editor philosophy. I had to consider three things in the design: the way the editor will work internally; how it can be customized; and what word-processing features it will include.

Internal Text Formats and Operation

I decided to adapt the window text editor from my books about Turbo C and QuickC. The code works properly, is mine and yours to use, and is reasonably small. It is modified here to use the more concise window functions from this column and to be extensible in that it is easy to add commands without modifying the source code of the editor itself. It also now has the potential for recursive calls to the editor. Although there are no immediate plans for a multiple-window editor or for one that can be called recursively, the editor should not preclude those features. Therefore, the code will allow an application to preserve the editor's current environment and reinvoke the editor for a different environment (window, buffer, and so forth).

To use the window text editor in a program, you must provide a window

(continued from page 104)

and a buffer with, perhaps, some text already in it. The buffer's size is a function of the width of the widest allowable text line and the maximum number of lines of text. You tell the editor these two dimensions when you execute it. The editor assumes that the window is wide enough to hold the lines and that the buffer is deep enough for a text file with the maximum number of lines. This approach, which uses trailing spaces to fill out each line, uses more buffer memory than one that uses newlines in the text, but the code for managing the cursor and blocks of text is less complex. The buffer is essentially an *xy* rectangle of *y* rows by *x* columns.

Because the fixed rectangular buffer must fit horizontally inside the window, the editor does not provide for horizontal scrolling. This means that lines cannot extend beyond the right margin of the window. It also means that a text file cannot have variable margins. These are features that we trade off to get an efficient editor within an application, and their loss does not compromise our requirements for a message composition tool.

The editor does not record or otherwise use the tab character (\ t) in a text buffer. When you type the Tab key the editor inserts the appropriate number of spaces to put the cursor at the next tab stop. Tab stops are specified as fixed intervals of character positions. The width of the interval can be configured when you compile the editor code.

The editor allows you to mark text blocks for move, copy, delete, and paragraph operations. These blocks are online boundaries rather than at character positions. This choice, too, was made in the interest of efficient code.

Customizing the Editor

Customizing the window editor consists of changing some default values and providing your own command keys instead of the ones that are published. If you make such changes, when the menu shell and help windows are added later you will need to change the text that tells you what these commands are. For now, however, you only need to redefine the configured command values found in editor.h. The editor commands are single key stroke values that are delivered by the function *getkey* in window.c (September *DDJ*). The configurable commands have mnemonic global names such as END_LINE and DELETE_WORD. Change their values by defining different values for them. Note that this approach uses single keystrokes as editor commands. If you want to make this editor emulate the WordStar double stroke sequences like the Borland and QuickC editors do, you must write a *getkey* substitution that knows about them. There is a precedent for doing this —lots of people are comfortable with the WordStar command set —but the editor as published here does not support double stroke sequences.

Three other configuration items are found in editor.h. These are the width of the tab stops, whether the editor comes up in insert or overwrite mode, and whether the editor comes up with automatic paragraph reforming enabled.

To change the editor's window colors, change the TEXTFG, TEXTBG, BLOCKFG, and BLOCKBG global values in window.h from the September "C Programming" column. Text colors are the normal colors for the text editor window and should be specified as the window's colors when you establish the window that the editor will use. Block colors are set by the editor when a block is marked. The editor resets the window to the text colors for nonblocked displays. Color configuration was discussed in September.

Of course, one other level of customization is available to you. You have the source code to this editor, and you can make it do anything you want within the limits of your abilities with the C language.

Word Processing Features

No one will use this editor to replace Word, WordPerfect, XyWrite, or WordStar; nor will you toss out Brief, vi, Vedit, EMACS, or the Norton Editor. Nonetheless, the editor needs a sufficient set of word processing features to make it useable for entering messages. From my experience with word processors and on-line services comes a set of features that I believe to be the minimum, and these are built into the editor. Realizing, however, that the need for more features will no doubt come later, I wrote the code so that extensions —those new, undiscovered features —could be easily tacked on. More about that later; first the minimum features.

Text Entry —You can type words into the buffer. Word wrap occurs when the word being typed reaches the right margin or when an inserted character pushes the rightmost word to the margin. When you type, the insert or overstrike mode is effective. Paragraphs can be automatically reformatted as you type or not as determined by a programmed toggle.

Cursor Movement —You can move the cursor around by characters, words, pages, tabs, to the right or left of the line, to the top or bottom of the window, or to the beginning or end of the text. If you type the Enter key, the cursor moves to the next line, first column. If this is done in insert mode, the line is split.

Editing —You can delete characters (left or right), words, lines, or blocks. You can mark a block of lines and then move, copy, delete, or form a paragraph from the block. You can unmark a marked block.

Paragraphs —A paragraph begins with an indented line. The indent is the number of spaces in the tab interval. Automatic paragraph reforming occurs from the cursor position up to the next paragraph or blank line. The paragraph command does likewise when no block has been marked.

These are the basic text editing features of a buffer of text that is edited through a window by our text editor. The file management aspects of the editor project come next month when we see how to read and write text files into and from the buffer. We will also add the text searching algorithms and a menu shell. Nothing about this program pushes the technology. We have added a tool to our collection, one that will enable free-format text to be entered and changed in a window by the users of our programs.

The Editor Software

Listing One is editor.h. It is used to define the editor's command set and default mode settings. The *#define* statements for the commands assign key values to command mnemonics. The key values are taken from window.h or are defined here. Keystrokes have the values returned by the *getkey* function in window.c. The values for function keys are formed by adding the value 128 to the scan code returned by BIOS thus setting the most significant bit and forming a unique 8-bit value for the key stroke. Because window.h does not have all the possible key values defined, each addition to the tool set that needs other key values must define them themselves. Thus, we have the two Alt key definitions at the top of editor.h.

The TAB mnemonic defines the width of tab stops. As published here, TAB is 4, so tabs will occur at 5, 9, 13, and so on. If you change TAB, the tab posi-

tions will change accordingly.

The REFORMING variable is set to TRUE or FALSE to specify whether automatic paragraph reforming will occur as you type. If you set it to TRUE, the editor tests to see if the paragraph needs to be reformed each time a character or word is deleted and when word wrap occurs. This test compares the white space at the end of the current line with the length of the first word on the next line. If the word will fit at the end of the current line, the paragraph is reformed. This is a convenience when you are entering raw text; it would be a pain in the neck for code or a table. Therefore, the menu software next month will include a command to turn the mode on and off. On some slow processors the automatic reformat gets sluggish when a word is wrapped near the top of a long paragraph, and the display of the keys you type falls behind the speed of a medium to fast typist. To improve this performance, remove the call to *test_para* in the function *carttn* in editor.c. This will change the reformat rules during word wrap to work on just the current and next line of text, pushing the rest of the paragraph down a full line when room is needed. Later a press of the PARAGRAPH command (F2) will complete the reformat operation. This approach is close to the way WordStar works. The original approach is similar to, but not as fast as, XyWrite.

The INSERT variable is set to TRUE or FALSE to indicate whether typing is in insert or overstrike mode. The value assigned to the global symbol is the default mode when the editor is started. Thereafter the Ins key toggles the mode. The *#ifndef* is for programs that use the data entry functions from October. That library also has an INSERT mode for data entry templates, and the *#ifndef* prevents the two *#define* statements from colliding.

editor.h defines the *edit_env* structure, which contains all the variables related to the environment of a particular invocation of the editor. Later, if we decide to use multiple windows or if we need to invoke a secondary editor from within a primary one, we will stack the incidence of the structure declared as *ev* in editor.c.

editor.c contains the window text editor function. You call the function named *text_editor* and pass it the address of your edit buffer, the maximum number of lines in the buffer, and the length of the longest line. You must have established a window with *establish_window* in window.c, and that window must be able to contain lines of the width specified in the call to *text_editor*. In other words, the window must be at least as wide as the line length plus two for the window borders. The size of the buffer must be at least the line length times the line count and should contain displayable text data or spaces. The *text_editor* function returns the keystroke that terminated the edit session. This value will be either the Esc key or the QUIT command (Alt-Q as published). A program can test this value to know what the user intends to do with the buffer of text.

C language code is not always its own best documentation. It is, however, the most reliable statement of what is going on in the program.

The *text_editor* function displays the text in the window and begins accepting data keys or commands from the user. With each keystroke the function pointed to by the *status_line* function pointer is called. This allows the application program to show buffer status information such as the page and line numbers and the mode settings. To use this feature, the application must initialize the pointer to the address of the function that displays the status. We will use this feature next month.

The variable named *forcechar* appears on lines 84 and 85. If *forcechar* has a non-zero value, that value is substituted for the next key press. This mechanism allows external code to force the execution of a command. External code can be defined by an address in the function pointer named *editfunc*. If you initialize this pointer to the address of a function, the function will be called whenever the editor cannot recognize a keystroke. The value of the keystroke is passed in the call to the function on line 244. The function can view the external structure named *ev* to examine the editor's environment, it can modify that environment, and it can force execution of a command when it returns by placing a

command value in *forcechar*. This mechanism will be used next month to add file management, menus, and text searching to the editor. It is how we make the editor extensible without modifying the code in the editor itself.

Crotchet Number Six: Obsolete Comments

Most of the code in editor.c explains itself about as well as C code can explain itself without extensive comments. At least I think so. It works for me and I hope it does for you too. My commenting practices follow a convention that identifies the purpose of each function in a comment at the beginning of the function. Variables that are not obvious get comments that describe their purpose. Where code gets downright abstruse, I will insert comments as it goes along, but mostly I prefer to let the code describe itself. This habit and preference comes from years of reading the code of others where their extensive and verbose comments predate the code by generations of modifications. I have been lulled into believing beautifully crafted comments and have thus been subliminally conditioned to assume things that are not true. This loses time. Later, when my confusion reaches an intolerable level, I resort to reading the code, only to find that it disagrees with the comments. No matter whether the comments are statements of intent never realized or accurate descriptions of code no longer in place; the comments say one thing and the code does another.

As a matter of conviction and to preserve the remnants of my sanity, I now refuse to read comments that are written as pseudocode unless I am sure the program has never been modified. Even then I am reluctant. Those comments are rarely (if ever) maintained as the code is being developed or later when it is changed. Perhaps your experiences are different; perhaps the rigidly enforced standards and procedures of your employer keep this crotchet out of your shop; and perhaps you believe that. I do not.

C language code is not always its own best documentation. It is, however, the most reliable statement of what is going on in the program. My practice serves me well because I tend to remember what those cryptic variable names mean, and I prefer small functions with simple purposes. If the function works, its purpose is understood, and it is small, then it can be thought of as a black box, can be read, and can be trusted. Not that I always practice what I preach; *text_editor* is a big function, although it is mostly a lot of small *case*s to a key stroke *switch*.

By the way —I don't much care to read code that has been commented out either. A faraway, out-of-sight #if DEBUG statement or an as yet unterminated /* start-of-comment token can have you reading reams of code that does not exist.

On the other hand, my pal Bill Chaney says that anyone who fails to provide ample comments in an assembly language program should be required for penance to spend a year maintaining COBOL/CICS screen driver programs for the Puerto Rican income tax system.

An Example for Using the Text Editor

Testedit.c, Listing Three, page 131, is a simple example of the use of the text editor. Compile and link testedit.c with editor.c and window.c. You run it by naming a text file on the command line. This is not a standard text file; it is an image of the editor's rectangular buffer, so the first time you run testedit, you can give a file name that does not exist and the program will build it. testedit establishes a window, reads the file —if it exists —into the buffer, and calls the *text_editor* function. If the Esc key is not returned, which means that you pressed the QUIT key (Alt-Q), the buffer is written to the file that you named on the command line. If you press Esc, the program exits without writing the buffer.

This program is not very smart. Its purpose is to demonstrate how to set up, use, and exit from the window text editor. Next month's offering incorporates this new editor engine into the tiny word processor I mentioned earlier. I will take drastic measures to test the tiny word processor when I write next month's column. I will, temporarily at least, abandon my beloved XyWrite and use the tiny word processor for the month's work, risking the fruits of my creative labors to a new and unproven text editor. This, dear readers, is dedication and commitment. Expect nothing less from a *DDJ* columnist.

Availability

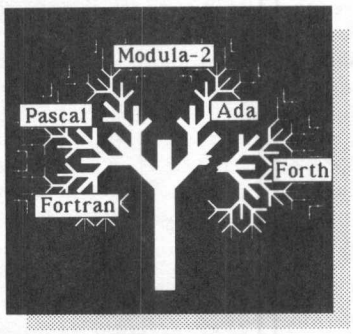
All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings start on page 119.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Examining Turbo Pascal 5.0



Now in its fifth year, Turbo Pascal remains the hottest name in the Pascal game, and the new Turbo Pascal 5.0 shows that Philippe and friends down Scotts Valley way intend to keep it there. TP5 introduces the much-rumored and long-awaited debugger and reintroduces overlays, plus a host of other user-demanded enhancements, and TP5 now comes in a couple of different pricing packages geared toward both casual and serious programmers.

Borland rolled out TP5 around Labor Day, along with Turbo C 2.0 and TASM, the new (and also much-rumored) Turbo Assembler. Unlike the other Borland language products, TASM has no integrated environment. It's strictly a command-line assembler, but it comes with a standalone debugger. The debuggers integrated with Turbo C and Turbo Pascal are identical to each other, and that's good news for those of us who switch around between languages. The freestanding debugger can also be used with high-level languages.

As for TP5, you still have your choice between the integrated and command-line compilers. The environment has been enhanced to accommodate the debugger, which is seamlessly folded in. In most other respects, the environment is pretty much the same: a little culture shock for those upgrading from TP4.

And what about a linker, you ask? Well, that's part of the compiler, as in the earlier release. No, Version 5.0 still doesn't produce .OBJ files, so you can't export Turbo Pascal object modules to other languages or systems. This is because both Versions 4.0 and 5.0 produce a proprietary object module format called a Turbo Pascal unit, which has the filename extension .TPU. You *can* import externally generated .OBJ files —for example, those produced by TASM —into TP5 programs, but you

by Kent Porter

can't export .TPU object modules to other language systems because they don't understand the format.

Borland contends that the .OBJ format is inherently inefficient and that the Turbo Pascal compiler owes its impressive speed to a more efficient object form. And indeed, the source-to-.EXE time for equivalent programs in TP5 and TC2 (the latter uses .OBJ files) is faster for Pascal. Still, .TPU files remain the only way to modularize Turbo Pascal programs, and there are sure to be grumbles.

Virtually all the rest of the news about TP5 is good. In addition to the big item —the debugger —there are a lot of smaller but telling improvements to the product. For example, the help subsystem has been expanded to include language help that's context-sensitive. Pressing F1 brings help for the environment state you're in or the menu selection currently highlighted, as before. For language help, you position the cursor on an occurrence of the identifier in source code, then press Ctrl-F1. A description of the language element in question appears in a pop-up screen with a brief description and an example. This is a whole lot easier than looking it up in the manual. Additionally, if a compile- or run-time error crops up, you can get a diagnosis of the probable cause(s) by pointing to it and pressing F1. This is help that's truly helpful.

Other enhancements include dead code and data removal, 8087 emulation so that you can take advantage of all REAL types even if the machine lacks a coprocessor, and constant cascading. The following was illegal in all previous releases of Turbo Pascal (and still is in most other Pascal implementations):

```
CONST BuffLength = 256;
      HalfLength = BuffLength DIV 2;
      DblLength = BuffLength * 2;
```

This C-like handling of constants is legal in TP5. Also C-like is TP5's new ability to handle procedural types, variables, and parameters, thus allowing you to pass, say, a function name as an argument to another subprogram and then, within that subprogram, invoke the parametric function via its variable name. The TP5 linker is also able to resolve circular unit references. This eliminates the irksome problems that TP4 sometimes produced when two or more units were heavily interdependent.

The awkwardly thick manual of TP4 has been divided into two with Version 5.0: a User's Guide and a Reference Guide. The 350-page User's Guide covers installation and use of the product. The Reference Guide devotes almost 500 pages to programming issues and a comprehensive description of the Turbo Pascal language. The built-in identifiers are arranged in alphabetic order, and nearly every entry includes a programming example.

Among other behind-the-scenes enhancements are support for EMS 3.2 or higher for editing and debugging large files, an expanded graphics library, and a word- or byte-alignment option (word alignment yields better .EXE execution time).

Borland has added an INSTALL program with options for floppy and hard-disk installation. The hard-disk setup also lets you choose between a first-time install and an upgrade from TP4. There are scads of new examples and also some documentation text files that come in compressed format along with an unpacking utility; you *can choose* whether or not to explode the compressed files at installation time. INSTALL recommends a default subdirectory structure for various kinds of modules (.PAS, .TPU, .EXE, and so on). This is good for those of us who were too lazy to set up our own under earlier releases. I must have 400 files in my old \TP directory, which is absurd, but I'll bet I'm not the only one. The default structure helps one manage files more sensibly.

Now let's get to the really good stuff.

Seamless Debugging

Interactive debugging is an integral part of the TP5 environment. Unlike Microsoft's CodeView, which is a separate utility, the Turbo debugger is built in. You know it's there because there are new keystrokes to learn and because some of the menus have new selections that pertain to debugging. You don't have to exit to the debugger from the Turbo Pascal or Turbo C environment as you do in order to use CodeView.

This makes debugging a natural part of the program development process

(continued from page 110)

rather than an extension of it, and surely that's how the gods of computing intended for it to be. You can set breakpoints, test, edit, recompile, watch variables, change them, or step through the source code watching it execute and ironing out the wrinkles.

An example of this seamless debugging is the sticky breakpoint. Let's say you set a breakpoint at line 253 in the source. As you work your way through the code, you discover a bug at line 194. You stop and edit, removing three lines, then recompile. The break at line 253 is still set, but since you've removed three lines, it's now at line 250, and execution will halt when it gets there. The breakpoint is sticky because it sticks with the source line to which it was originally attached, even if the line moves.

The integrated debugger adds about a dozen new command keystrokes to the environment, plus half a dozen menu selections without shortcuts. In working with it, I've found seven that are essential and that give some idea of the debugger's capabilities:

- Ctrl-F2 Reset program to starting condition
- Ctrl-F3 View the calling stack (by subprogram name)
- F4 Run from currently executing source line to cursor row
- Ctrl-F4 Evaluate expression or view/change variable
- Alt-F5 Toggle to the user screen
- F7 Single-step, tracing into subprograms
- F8 Single-step, stepping over subprograms

There are other shortcuts as well. Additionally, you can use selections from the Break/Watch menu (or Ctrl-F7) to set up a watch window that continuously displays the current value of one or more variables as the program runs. The watch window at the bottom of the display grows and shrinks dynamically according to the number of variables being watched at the moment. This ensures that you always have the maximum edit/debug window size. You can set up watch expressions using format specifiers that show, for example, monetary notation for REALs, hex format, and the layout of complex records.

Any time the debugger halts the program being debugged, the current source line (the one that will be executed next) is highlighted by a cyan background on the color monitor or by reverse video on a monochrome display. If you want to examine a variable, evaluate an expression, or even test a program function, you press Ctrl-F4. This pops up a three-part dialog box. When the cursor is on a variable name, the top box shows the identifier and the middle box shows its current value.

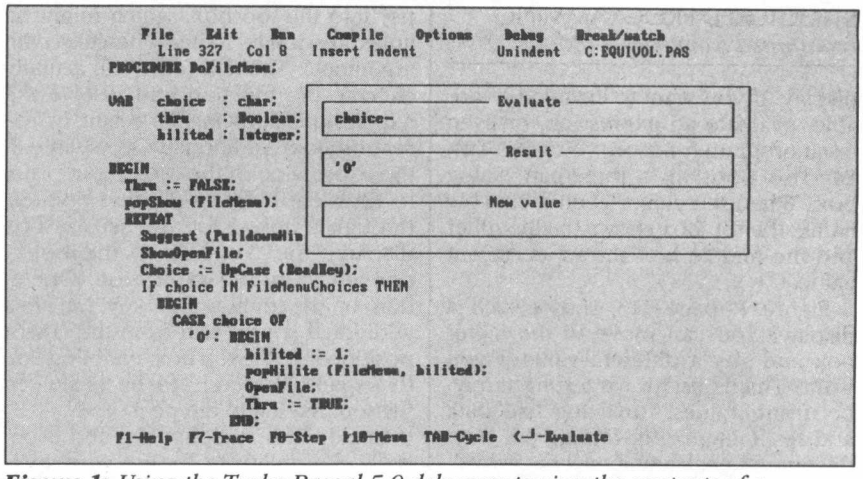


Figure 1: *Using the Turbo Pascal 5.0 debugger to view the contents of a variable while execution is halted at the IF statement*

Figure 1, page 113, shows such a display. You can move to the lower box and key a different value if you want. This is useful for testing rarely-occurring values, for fixing bad data and resuming, or for finding out how the program behaves if unexpected values occur. You can also type an identifier into the top box, which might be an expression including a function with arguments. The debugger will actually execute the function and resolve the expression, showing its result in the second box. Additionally, you can edit the expression in the top box in order to do such things as chasing pointers through a linked list; the second box displays the contents of the object pointed to by the expression. Operation of the right-arrow key transfers additional source text from the cursor position into the top box, enabling you to expand the code to be evaluated without having to retype it.

Capabilities such as this, and especially the ability to fix errors, recompile, and resume the session without disrupting the debugging setup, make the integrated Turbo debugger an unparalleled environment for program development. The only things one could wish for are a function key template — I keep a 3×5 card next to the keyboard as a cheat sheet —and mouse support.

You can also use the standalone debugger that comes with TASM and with the bundled Turbo Pascal 5.0 professional package to debug Pascal and mixed-language programs. The freestanding debugger has more capabilities than the one integrated into the environment, but it lacks interactive compiling/assembling and sticky breakpoints. In other words, it operates in a manner analogous to CodeView.

Version 5.0 redresses the grievance with dividends, because it not only restores overlays to Turbo Pascal but makes them smart

The other major improvement in TP5 is overlays.

Getting Smart About Overlays

In one respect, the earlier update from Turbo Pascal 3.0 to Version 4.0 represented a step backwards: TP3 supported overlays, while TP4 did not. Version 3.0 provided only .COM files with a maximum code size of 64K, making overlays a necessity for larger programs. Because Version 4.0 removed this restriction with .EXE files, making it possible to write code that occupied the entire available memory of the PC, Borland apparently assumed that the need for overlays had gone away. They were wrong, as a chorus of protest from Turbo Pascal programmers swiftly proved.

Version 5.0 redresses the grievance with dividends, because it not only restores overlays to Turbo Pascal but makes them smart. In TP3, the programmer had to manually isolate chunks of code, ensuring that any given over-

lay only called subroutines within itself and not subroutines within another overlay. This was because only one overlay was in residence at a time, loaded by the root module of the application. Therefore communication among overlays was possible only by passing information through global or common variables owned by the root, which held values deposited there by the departing overlay and picked up by the one newly loaded. This placed a heavy burden of responsibility on the programmer and made overlay programming the sanctum sanctorum of gurus.

Not so in TP5. It uses Least Recently Used (LRU) algorithms to manage unit-based overlays on a subroutine-by-subroutine basis, and additionally takes advantage of EMS (if present) to further reduce overlay swapping.

You must still decide if overlays are necessary, but this is usually apparent from the sheer amount of source code and certainly so if a developing program suddenly starts crashing due to out-of-memory conditions. In such cases, it's necessary to recompile units with the $O+ directive and to USE the TP5 Overlay unit in the root module. A certain amount of additional planning and programming is required. The TP5 Reference Guide covers the whole subject in six pages —about 20 percent of which is example code —that should serve as an indicator of relative difficulty. The compiler takes care of the rest, managing overlay memory, bringing in subroutines as called, and, if low on memory, placing them atop the least recently used subroutines. This technique often allows enormous programs to execute at a speed close to that of a machine with unlimited memory.

Turbo Pascal 5.0 thus overcomes the major complaints about TP4 —no debugging and no overlays —while enhancing the package with numerous new features. All this has a price.

Redressing the Package

The lackluster packaging of TP3 and the dark, brooding colors of TP4 have been replaced in TP5 with yellow. A brighter color scheme reflects the livelier product inside; it might also serve to deflect attention from the price increase. With TP5, the base price rises to $149.95.

Sigh. Like many of us, I bought my first copy of TP2 —list price $49.95 — from a discounter for about $35. A first-timer will now have to spend three or four times that amount from the same source.

On the other hand, look how much more he or she will get: interactive debugging, .EXE programs of unlimited size, highly advanced graphics, some 230 built-in procedures and functions, and a source-to-executable performance increase of at least tenfold since Version 2.0. The price may be going up, but the deliverables are still outrunning it. Turbo Pascal remains a bargain.

Borland's new Professional package is an even better buy for the truly serious programmer. For $250, you get the base-line Turbo Pascal 5.0 compiler, plus TASM, and the standalone debugger. TASM supports all the Microsoft MASM dialects (along with their infamous inconsistencies) via metacommands, along with its own "ideal" syntax that assembles several times faster than MASM. TASM and the debugger sell by themselves for $150.

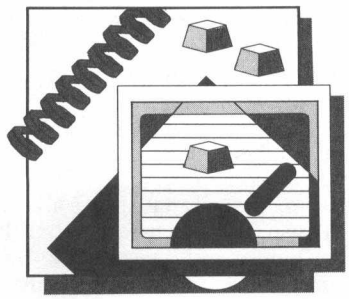
The math isn't hard to work out. TP5 + TASM = $300, priced at $250. If you bought TP4, listed at $100, you can upgrade to the Pro pack for another hundred bucks: several times the development ease and software at a fraction of the price. For those not interested in the assembler, a straight upgrade from TP4 to TP5 is $49.95.

Go for it. As a jaded reviewer, I don't impress easily, but TP5 is a superlative package for Pascal programmers.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Programming Languages as Chinese



The motivation for this column is a sentence from Robert Floyd's Turing Award lecture, "The Paradigms of Programming:" "I believe that the best chance we have to improve the general practice of programming is to attend to our paradigms." Programming paradigms —the broad approaches to problem solving, the sets of unwritten rules accepted by communities of programmers, the model problems that condition our thinking about a problem well before we get to the point of selecting an appropriate algorithm —can enslave or liberate our thinking about our craft.

The column's premise is that by exploring alternative paradigms, we can free ourselves from the mental chains of the widely accepted and unquestioned paradigms. That means more questioning than answering, and it means looking at some futuristic approaches to software development. This exploration of the not-yet-but-soon-to-be-practical is fraught with difficulties: finding the substance under the hype in writings on artificial intelligence, for example, is one of the model problems of the technical writer's paradigm.

What Will You Get Mom for Christmas?

A particularly lurid example of AI hype crossed my desk recently in the form of a press release for a new book. You will no doubt forgive me if I don't embarrass the author/publisher by naming the work, particularly since, having burdened the brief book with an 11-word, 6-buzzword title, he has already named it more than adequately. The book purports to explain thinking and artificial intelligence, lay out a new cognitive model, and completely solve the problem of the management of complexity. Here are some samples of the release's claims: "Next, the author reexamines the nature of reality itself . . . explains how the human brain works . . . a computer can be made to 'think' and create 'new' thoughts . . . by Christmas [you will be able to] experience your own 'thinking computer.' "

by Michael Swaine

Nowhere in the press material is the book connected with any other work on AI or thinking, except for dropping the names of Francis Bacon, Rene Descartes, and George Boole.

A lot of the promotional materials for expert system implementations are almost this bad. That's why I've not done much on expert systems so far. But the intended scope of this column includes expert systems and is even broader than that; I want to examine the paradigmatic approach itself. For example, it's not out of order here to discuss what programmers have to teach writers about HyperText. Writers need to expand their paradigms, too.

What Can Programmers Teach Writers about HyperText?

A recent article on the promise of HyperText claimed that, with the advent of HyperCard and other tools for flexible organization of text, the author faces a trade-off between reader control and author control of the structure of the material. Some links between elements of a work must be followed sequentially if the reader is to follow the argument or the story line. Other links may be followed in any order without penalty. Traditional hard-copy writing stands at one end of the spectrum of control and the electronic version of William Burroughs-style cut-and-randomly-paste organization stands at the other end, with the reader wielding the scissors and paste bottle. The problem is to pick the right point on the spectrum for your particular subject matter. Yeah, identifying the problem is always useful, but the writer of the piece I was reading didn't seem to have a clue to the solution. I think programmers have all the clues.

The HyperTextual author needs to deal with many of the issues that programmers have to deal with in writing structured or object-oriented code. Modules need to be defined at the right level. Each should serve one purpose. The logical structure should be mapped out explicitly and the actual structure should mirror it. Connections not in the logical structure should not appear in the actual structure. Some components may need to be nested within others.

These principles, familiar to programmers, can actually be applied to writing when one is not tied to the linearity of the page, but no one is really doing it yet, and no one knows what kind of writing will result when writers work this way. It seems clear that it won't much affect traditional forms, since they are, well, traditional forms. Telling a story is a sequential process by definition. It will remain a sequential process, and people will continue to tell stories. Explaining how to perform a complicated function may or may not be entirely sequential, but its sequential aspects will continue to be dealt with sequentially. Reference books do not need to be presented sequentially and are the most obvious beneficiaries of nonsequential writing. But new forms will emerge as well.

It looks to me like it's neither intrinsi-

cally hard nor easy to write this way, any more than writing a reference book is intrinsically harder or easier than writing a novel. It's just a different paradigm for writers, and I'm betting that writers who also happen to be programmers will get it right first.

I must admit that I worked through that little exercise in programmer backpatting to soften you up for a new look at the point-and-shoot paradigm of programming.

What Makes Scott Watson Twitch?

If you're like most sensitive software developers, your reaction to a point-and-shoot icon-based programming language is probably something like Scott Watson's.

At the MacWorld Expo in Boston this summer, several developers, including Macintosh system-software developer Andy Herzfeld and Scott Watson, the author of *Red Ryder*, sat on a panel and fielded questions from the audience. A typical question was: What are you working on right now? Andy Herzfeld fielded that one, saying that he couldn't really talk about the new hardware that he and Burrell Smith were working on, but that he had found it necessary to write a new language to support that effort, a completely wordless language that uses no keyboard. He does all his programming these days, he said, with a mouse, by pointing at icons.

Scott Watson twitched convulsively.

It was a magnificently expressive act of communication, and it brought down the house. In one gesture, he got across his abject horror at the very idea of a nonverbal programming language and communicated it nonverbally so effectively that he broke up the audience. I'm a pretty good writer and I told the story as well as I could, but you probably didn't fall on the floor laughing, right? You really had to see it.

End of anecdote. The argument against picture languages is more than anecdotal. George Morrow has been explaining for some time why iconic languages like Chinese lack the expressive power of character-based languages like English, and how this applies to programming. For purposes of argument I am going to simplify Morrow's position probably beyond the point where he would like to be associated with it today. What I will be presenting is a fairly common attitude toward iconic languages, the attitude touched when Scott Watson twitched, and an attitude that George Morrow has in the past pretty explicitly espoused. I'll call it Simple Morrow.

Here it is: Pictures are a poor and an imprecise tool for communicating abstract ideas. Pictures are static, they have no obvious rules of combination comparable to the rules for combining words in character-based natural and programming languages, and a picture vocabulary requires many more elementary symbols than the 26 letters of the Roman alphabet or the 256 characters of 8-bit ASCII. If pictures are a good basis for a language, why are there no purely iconic languages today? Hieroglyphics never changed and died out, while Chinese has evolved into a partly-phonetic, partly-symbolic menace to communication. Icons are for people who can't read.

Will Your Grandchildren Speak Chinese?

In an excellent new magazine called *Language Technology*, I recently came across a strong counter proposal. The article "Will Your Grandchildren Speak Chinese?" reports on a word processing/translating system that allows a user to type Pinyin (Roman character text) Mandarin Chinese and converts it to Hanzi ideographs faster than one can type the same text in English. The author, who is chairperson of the Machine Translation Committee of the New York Circle of Translators, wraps up the piece by speculating that Chinese could replace English as the world's common language, precisely because Chinese is a better language than English for writing computer programs. You might want to read that sentence again. He asks, "Could it be that the developers of computer languages have been struggling to reinvent Chinese?"

The piece was not very technical and its premise not, I think, very defensible; but what I think is interesting is that an intelligent person, looking at the issues and the evidence, could put forth such an argument. I think that he can do so because icons are more complex than Simple Morrow thinks.

How Long Is a Piece of String?

Are iconic programming languages a bad idea? And if they do have merit, are they only of value to people who already read an iconic language like Chinese?

There are at least three levels of abstraction possible when using iconic or visual encoding. Pictures look like what they represent, abstract symbols bear some analogical relationship to their referents, and arbitrary signs need have no relationship to what they represent.

If both icons and strings of characters are used arbitrarily, their relative information content is a simple mathematical issue. How many pixels make up an icon, and how many bits per pixel? How long can the string be? It seems that some concepts, such as the existence operator, require such arbitrary representation from either an iconic or a character-based language.

The mapping from strings of characters to concepts in a programming language or a natural language is at the lowest level entirely arbitrary. That would seem to suggest that adding some meaningfulness to the mapping when possible, as Chinese does by retaining some representational ideographs, could only increase the communicative ability of the language. That a programming language that uses visual information when it's useful but allows for combining icons into new icons and permits the arbitrary association of icons with concepts —that such a language would be more expressive and easier to use than a character-based language. And that may be the case for those used to thinking in Chinese.

Chinese gets a bad rap from Simple Morrow. I can't read Chinese, but I understand that it is a very expressive language, with a combination of abstract symbols and arbitrary signs combined according to rules as powerful as the grammar of English. It supports subtle philosophical discussions, deep and beautiful poetry, and may even permit the representation of programming concepts. Its biggest drawback seems to be not the expressive power of its symbols or the difficulty of reading them, but the difficulty of producing them. They require a very big keyboard. And there seem to be solutions to that problem, from using a character-based front end (which is not exactly an admission of the superiority representational power of character-based languages) to Andy Herzfeld's keyboardless approach.

Herzfeld is evidence that at least one native speaker of a character-based language has found a reason to use, even to create, an iconic language. But it does seem clear that we character types need at least to be able to use something like our natural language for things like naming variables. The Roman alphabet is too powerful a tool not to use. Some icons, of course, include text. Perhaps the ideal approach is a model in which every entity has both a character-string name and visual attributes, each of which might be inherited in full or part from its class. Then you could program using icons or text or a mixture of the two.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Listing One *(Text begins on page 12.)*

```
#include <string.h>

int stcknum;
char *ststr1l[26];
char *ststr2l[26];
char *ststr1r[26];
char *ststr2r[26];

int simil (str1, str2)      /*'Pattern Matching by Gestalt' */
char *str1, *str2;          /*by John W. Ratcliff           */
                            /*Dr. Dobb's Journal #141.      */
{                           /*July 1988                     */
    int len1;
    int len2;
    int ncmp;
    int score;
    char *di;
    char *si;
    char *de;
    char *se;
    char *cl1;
    char *cl2;
    char *cr1;
    char *cr2;

    score = 0;
    stcknum = 0;

    len1 = strlen(str1);
    len2 = strlen(str2);
    if (len1 == 0 || len2 == 0)
        return (score);

    pushst (str1, str1+len1-1, str2, str2+len2-1);

    while (stcknum != 0)
    {
        popst (&si, &se, &di, &de);
        cl1 = si;
        cl2 = di;
        cr1 = se;
        cr2 = de;
        if ((ncmp=compare (&si, &se, &di, &de)) !=0)
        {
            score += ncmp*2;
            if (cl1 != si    && cl2 != di   &&
               (cl1 != si-1  || cl2 != di-1))
               pushst (cl1, si-1, cl2, di-1);
            if (se    != cr1 && de    != cr2 &&
               (se+1  != cr1 || de+1  != cr2))
               pushst   (se+1, cr1, de+1, cr2);
        }
    }
    return  (100*score/(len1+len2));
}

int compare (si, se, di, de)
char **si;
char **se;
char **di;
char **de;
{
  int maxchars;
  int l;
  int len1;
  char *i;
  char *j;
  char *m;
  char *n;
  char *s2end;
  char *cl1;
  char *cl2;
  char *cr1;
  char *cr2;

  maxchars = 0;
  for (i=(*si); i <= *se-maxchars; i++)
  {
      len1 + *se - i;
      for (j=(*di); j <= *de-maxchars; j++)
      {
         s2end = j + len1;
         if (s2end > *de)
             s2end = *de;
         for (m=i,n=j,l=0;  *m==*n && n <=s2end; m++,n++)
             l++;
         if (l > 0)
         {
            if (l <= maxchars)
            {
                j += l-1;
            }
            else
            {
                cl1 = i;
                cl2 = j;

                  maxchars = l;
                  l--;
                  j += l;
                  cr2 = j;
                  cr1 = i+1;
               }
            }
         }
  }
  *si = cl1;
  *se = cr1;
  *di = cl2;
  *de = cr2;
  return (maxchars);
}

pushst (si, se, di, de)
char *si;
char *se;
char *di;
char *de;
{
   ststr1l[stcknum] = si;
   ststr1r[stcknum] = se;
   ststr2l[stcknum] = di;
   ststr2r[stcknum] = de;
   stcknum++;
}

popst (si, se, di, de)
char **si;
char **se;
char **di;
char **de;
{
    stcknum--;
    *si = ststr1l[stcknum];
    *se = ststr1r[stcknum];
    *di = ststr2l[stcknum];
    *de = ststr2r[stcknum];
}
```

End Listing One

Listing Two

```
int simil (s1, s2)
/* Ratcliff/Obershelp Pattern Matching */
char *s1, *s2;
{
  short l1, l2;

  l1 = strlen(s1);
  l2 = strlen(s2);

  if (l1 == 1)                    /* check for the easiest case */
    if (l2 == 1)
      if (*s1 == *s2)
        return(100);

  return(200 * GCsubstr(s1, s1 + l1, s2, s2 + l2) / (l1 + l2));
}

int GCsubstr(st1, end1, st2, end2)
/* recursive greatest common sub-string */
char *st1, *end1, *st2, *end2;
{
  register char *a1, *b1, *s1, *a2, *b2, *s2;
  short max, i;

  if (end1 <= st1)                /* s1 empty */
    return (0);
  if (end2 <= st2)                /* s2 empty */
    return (0);
  if (end1 == st1 + 1)            /* s1 has one char, */
    if (end2 == st2 + 1)          /* and s2 has one char. */
      return(0);                  /* They cannot be equal. */

  max = 0;
  b1 = end1; b2 = end2;

  for (a1 = st1; a1 < b1; a1++)
    for (a2 = st2; a2 < b2; a2++)
      if (*a1 == *a2)
        {                 /* How long is the common sub-string? */
        for (i = 1; a1[i] && (a1[i] == a2[i]); i++)
          ;
        if (i > max)
          {
          max = i; s1 = a1; s2 = a2;
          b1 = end1 - max; b2 = end2 - max;
          }
        }
  if (! max)
    return (0);

  max += GCsubstr(s1 + max, end1, s2 + max, end2);        /* RHS */
  max += GCsubstr(st1, s1, st2, s2);                      /* LHS */

  return(max);
}
```

End Listings

Listing One *(Text begins on page 104.)*

```
/* ------------ editor.h ------------- */
#define ALT_Q 144
#define ALT_R 147
/* -------- configured editor commands ---------- */
#define BACKTAB         SHIFT_HT
#define NEXTWORD        CTRL_FWD
#define PREVWORD        CTRL_BS
#define TOPSCREEN       CTRL_T
#define BOTSCREEN       CTRL_B
#define BEGIN_BUFFER    CTRL_HOME
#define END_BUFFER      CTRL_END
#define BEGIN_LINE      HOME
#define END_LINE        END
#define DELETE_LINE     ALT_D
#define DELETE_WORD     CTRL_D
#define INSERT          INS
#define QUIT            ALT_Q
#define PARAGRAPH       F2
#define BEGIN_BLOCK     F5
#define END_BLOCK       F6
#define MOVE_BLOCK      F3
#define COPY_BLOCK      F4
#define DELETE_BLOCK    F8
#define HIDE_BLOCK      F9
#define REPAINT         ALT_R
/* ------- configured default modes ----------- */
#define TAB 4
#define REFORMING TRUE    /* auto paragraph reformat mode */
#ifndef INSERTING
#define INSERTING TRUE    /* insert/overwrite mode        */
#endif
/* ---------- editor prototype ---------------- */
int text_editor(char *, int, int);
/* ------- macros ------------ */
#define curr(x,y) (ev.bfptr+(y)*ev.wwd+(x))
#define lineno(y) ((unsigned)(ev.bfptr-ev.topptr)/ev.wwd+(y))
/* ---------- editor environment ------------- */
struct edit_env    {
    int envinuse;       /* TRUE if the env is in use  */
    struct wn *wdo;     /* the editor window          */
    int wwd;            /* width of edit window       */
    int wsz;            /* size (chars) of window     */
    char *topptr;       /* -> first char in buffer    */
    char *bfptr;        /* -> first char in window    */
    char *nowptr;       /* -> current char in buffer  */
    char *lstptr;       /* -> last nonblank char      */
    char *endptr;       /* -> last char in buffer     */
    int text_changed;   /* TRUE if text has changed   */
    int nolines;        /* no. of lines in buffer     */
    int blkbeg;         /* marked block: 1st line     */
    int blkend;         /* marked block: last line    */
    int curr_x, curr_y; /* current buffer coordinates */
    int edinsert;       /* toggled insert mode        */
    int reforming;      /* toggled para reform mode   */
};
```

End Listing One

Listing Two

```
/* ---------------------- editor.c ---------------------- */

#include <stdio.h>
#include <ctype.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <mem.h>
#include <ctype.h>
#include "window.h"
#include "editor.h"

#define NEXTTAB (TAB-(ev.curr_x%TAB))
#define LASTTAB ((ev.wwd/TAB)*TAB)
#define PREVTAB (((ev.curr_x-1)%TAB)+1)

struct edit_env ev;         /* the editor environment      */
int do_display_text = TRUE; /* turns display on/off        */
extern struct wn wkw;       /* the current window          */
int forcechar;              /* externally force a kb char  */
void (*status_line)(void);  /* called once each keystroke  */
void (*editfunc)(int);      /* for unknown keystrokes      */

/* ---------- local function prototypes ----------- */
static int lastword(void);
static void last_char(void);
static void test_para(int);
static int trailing_spaces(int);
static int first_wordlen(int);
static int last_wordlen(void);
static void paraform(int);
static int blankline(int);
static void delete_word(void);
static void delete_line(void);
static void delete_block(void);
static void mvblock(int);
static void carrtn(int);
static void backspace(void);
static void fore_word(void);
static int spaceup(void);
static void back_word(void);
static int spacedn(void);
static void forward(void);
```

C PROGRAMMING

Listing Two *(Listing continued, text begins on page 104.)*

```
static int downward(void);
static void upward(void);
static void display_text(int);
static void disp_line(int y);
static void findlast(void);

/* ----- Process text entry for a window. ---- */
int text_editor(char *bf, int editlines, int editwidth)
{
    int depart, i, c;
    int svx, svlw, tx, tabctr, wraplen;

    current_window();
    depart = FALSE;
    tabctr = 0;
    if (ev.envinuse == FALSE)    {
        ev.wdo = &wkw;
        ev.wwd = editwidth;
        ev.wsz = ev.wwd * ev.wdo->ht;
        ev.topptr = ev.bfptr = bf;
        ev.nolines = editlines;
        ev.endptr = bf + ev.wwd * ev.nolines;
        ev.edinsert  = INSERTING;
        ev.reforming = REFORMING;
        ev.envinuse = TRUE;
    }
    set_cursor_type(ev.edinsert ? 0x0106 : 0x0607);
    display_text(0);
    findlast();
    /* ------- read text/command from the keyboard ------ */
    while (depart == FALSE)    {
        ev.nowptr = curr(ev.curr_x, ev.curr_y);
        if (status_line)
            (*status_line)(); /* external status line func */
        gotoxy(ev.curr_x + 2, ev.curr_y + 2);
        if (tabctr)    {          /* expand typed tabs */
            --tabctr;
            c = ' ';
        }
        else
            c = forcechar ? forcechar : getkey();
        forcechar = 0;
        switch (c)    {
/* ------------ fixed editor commands ----------------- */
            case '\r':
                carrtn(ev.edinsert);
                break;
            case UP:
                upward();
                break;
            case DN:
                downward();
                break;
            case FWD:
                forward();
                break;
            case '\b':
            case BS:
                if (!(ev.curr_x || ev.curr_y))
                    break;
                backspace();
                if (ev.curr_x == ev.wwd - 1)
                    last_char();
                if (c == BS)
                    break;
                ev.nowptr = curr(ev.curr_x, ev.curr_y);
            case DEL:
                movmem(ev.nowptr+1,ev.nowptr,
                    ev.wwd-1-ev.curr_x);
                *(ev.nowptr+ev.wwd-1-ev.curr_x) = ' ';
                disp_line(ev.curr_y);
                test_para(ev.curr_x+1);
                ev.text_changed = TRUE;
                break;
            case PGUP:
                ev.curr_y = 0;
                do_display_text = FALSE;
                for (i = 0; i < ev.wdo->ht; i++)
                    upward();
                do_display_text = TRUE;
                display_text(0);
                break;
            case PGDN:
                ev.curr_y = ev.wdo->ht-1;
                do_display_text = FALSE;
                for (i = 0; i < ev.wdo->ht; i++)
                    downward();
                do_display_text = TRUE;
                display_text(0);
                ev.curr_y = 0;
                break;
            case '\t':
                if (ev.curr_x + NEXTTAB < ev.wwd)    {
                    if (ev.edinsert)
                        tabctr = NEXTTAB;
                    else
                        ev.curr_x += NEXTTAB;
                }
                else
                    carrtn(ev.edinsert);
                break;
```

(continued on page 122)

Listing Two *(Listing continued, text begins on page 104.)*

```
/* --- displayable char: put in buffer --- */
if (ev.nowptr == ev.endptr-1 ||
   (lineno(ev.curr_y)+1 >=
       ev.nolines && ev.edinsert &&
       *curr(ev.wwd-2, ev.curr_y) != ' '))  {
    error_message("End of buffer...");
    break;
}
if (ev.edinsert) /* --- if insert mode --- */
    movmem(ev.nowptr,ev.nowptr+1,
        ev.wwd-1-ev.curr_x);
if (ev.nowptr < ev.endptr)    {
    if (ev.nowptr >= ev.lstptr)
        ev.lstptr = ev.nowptr + 1;
    *ev.nowptr = (char) c; /* put in buff */
    disp_line(ev.curr_y);
}
if (ev.nowptr == curr(ev.wwd-1, ev.curr_y) &&
    c == ' ' && ev.edinsert)    {
    if (strncmp(curr(0,ev.curr_y+1),
          "        ",TAB) == 0)    {
        carrtn(TRUE);
        break;
    }
}
else if (ev.endptr &&
        *curr(ev.wwd-1, ev.curr_y) != ' ')    {
    /* ------- word wrap is needed ------- */
    ev.nowptr = curr(ev.wwd-1, ev.curr_y);
    svx = ev.curr_x;   /* save x vector */
    svlw = lastword(); /* last word on line?*/
    ev.curr_x = ev.wwd-1;
    if (*(ev.nowptr-1) != ' ')
        back_word();
    tx = ev.curr_x;
    wraplen = last_wordlen();
    if (trailing_spaces(ev.curr_y+1) <
            wraplen+2)
        carrtn(TRUE);
    else if (strncmp(curr(0,ev.curr_y+1),
            "        ",TAB) == 0)
        carrtn(TRUE);
    else    {
        ev.nowptr = curr(0, ev.curr_y+1);
        movmem(ev.nowptr,ev.nowptr+wraplen+1,
            ev.wwd-wraplen-1);
        setmem(ev.nowptr, wraplen+1, ' ');
        movmem(curr(ev.curr_x,ev.curr_y),
           ·ev.nowptr,wraplen);
        setmem(curr(ev.curr_x,ev.curr_y),

/* -------- configured editor commands ---------------- */
            case REPAINT:
                display_text(ev.curr_y);
                break;
            case BACKTAB:
                if (ev.curr_x < TAB)    {
                    upward();
                    ev.curr_x = LASTTAB;
                }
                else
                    ev.curr_x -= PREVTAB;
                break;
            case NEXTWORD:
                fore_word();
                break;
            case PREVWORD:
                back_word();
                break;
            case BOTSCREEN:
                ev.curr_y = ev.wdo->ht - 1;
                break;
            case TOPSCREEN:
                ev.curr_y = 0;
                break;
            case BEGIN_BUFFER:
                ev.curr_x = ev.curr_y = 0;
                ev.bfptr = ev.topptr;
                display_text(0);
                break;
            case END_BUFFER:
                do_display_text = FALSE;
                ev.curr_x = 0;
                while (downward())
                    if (curr(0,ev.curr_y) >= ev.lstptr)
                        break;
                do_display_text = TRUE;
                display_text(0);
                break;
            case BEGIN_LINE:
                ev.curr_x = 0;
                break;
            case END_LINE:
                last_char();
                break;
            case DELETE_LINE:
                delete_line();
                ev.text_changed = TRUE;
                break;
            case DELETE_WORD:
                delete_word();
                ev.text_changed = TRUE;
                test_para(ev.curr_x);
                break;
            case INSERT:
                ev.edinsert ^= TRUE;
                set_cursor_type(ev.edinsert ? 0x106 : 0x607);
                break;
            case ESC:
            case QUIT:
                depart = TRUE;
                break;
            case PARAGRAPH:
                paraform(0);
                ev.text_changed = TRUE;
                break;
            case BEGIN_BLOCK:
                ev.blkbeg = lineno(ev.curr_y) + 1;
                if (ev.blkbeg > ev.blkend)
                    ev.blkend = ev.blkbeg;
                display_text(0);
                break;
            case END_BLOCK:
                ev.blkend = lineno(ev.curr_y) + 1;
                if (ev.blkend < ev.blkbeg)
                    ev.blkbeg = ev.blkend;
                display_text(0);
                break;
            case MOVE_BLOCK:
                mvblock(TRUE);
                ev.text_changed = TRUE;
                break;
            case COPY_BLOCK:
                mvblock(FALSE);
                ev.text_changed = TRUE;
                break;
            case DELETE_BLOCK:
                delete_block();
                ev.text_changed = TRUE;
                display_text(0);
                break;
            case HIDE_BLOCK:
                ev.blkbeg = ev.blkend = 0;
                display_text(0);
                break;
            default:
                if (!isprint(c))    {
                    /* ---- not recognized by editor --- */
                    if (editfunc)    {
                        /* --- extended commands --- */
                        (*editfunc)(c);
                        findlast();
                        display_text(0);
                    }
                    else
                        putch(BELL);
                    break;
                }
                            wraplen, ' ');
                        disp_line(ev.curr_y);
                        downward();
                        disp_line(ev.curr_y);
                    }
                    if (svlw)
                        ev.curr_x = svx-tx;
                    else
                        ev.curr_x = svx, --ev.curr_y;
                }
                forward();
                ev.text_changed = TRUE;
                break;
        }
    }
    return c;
}

/* ----- see if a word is the last word on the line ------ */
static int lastword()
{
    int x = ev.curr_x;
    char *bf = curr(ev.curr_x, ev.curr_y);
    while (x++ < ev.wwd-1)
        if (*bf++ == ' ')
            return 0;
    return 1;
}

/* --- go to last displayable character on the line --- */
static void last_char()
{
    char *bf = curr(0, ev.curr_y);
    ev.curr_x = ev.wwd-1;
    while (ev.curr_x && *(bf + ev.curr_x) == ' ')
        --ev.curr_x;
    if (ev.curr_x && ev.curr_x < ev.wwd - 1)
        ev.curr_x++;
}

/* ----- test to see if paragraph should be reformed ----- */
static void test_para(int x)
{
    int ts, fw;
    int svb, sve;

    if (ev.reforming && ev.curr_y < ev.nolines)    {
        ts = trailing_spaces(ev.curr_y);
        fw = first_wordlen(ev.curr_y+1);
        if (fw && ts > fw)    {
            svb = ev.blkbeg, sve = ev.blkend;
            ev.blkbeg = ev.blkend = 0;
            paraform(x);
            ev.blkbeg = svb, ev.blkend = sve;
            if (svb)
                display_text(0);
        }
    }
}

/* ---- count the trailing spaces on a line ----- */
static int trailing_spaces(int y)
{
    int x = ev.wwd-1, ct = 0;
    char *bf = curr(0, y);
```

```
        while (x >= 0)    {
            if (*(bf + x) != ' ')
                break;
            --x;
            ct++;
        }
        return ct;
    }

    /* ----- count the length of the first word on a line --- */
    static int first_wordlen(int y)
    {
        int ct = 0, x = 0;
        char *bf = curr(0, y);
        while (x < ev.wwd-1 && *bf == ' ')
            x++, bf++;
        while (x < ev.wwd-1 && *bf != ' ')
            ct++, x++, bf++;
        return ct;
    }

    /* ----- count the length of the last word on a line --- */
    static int last_wordlen()
    {
        int ct = 0, x = ev.wwd-1;
        char *bf = curr(x, ev.curr_y);
        while (x && *bf == ' ')
            --x, --bf;
        while (x && *bf != ' ')
            --x, --bf, ct++;
        return ct;
    }

    /* ------------ form a paragraph -------------- */
    static void paraform(int x)
    {
       char *cp1, *cp2, *cpend, *svcp;
       int x1, y1, firstline = TRUE;
       int y = ev.curr_y;

       if (!ev.blkbeg)    {    /* ---- if block not marked ---- */
           if (blankline(lineno(y)+1))
               return;        /* next line is blank, no reform */

        ev.blkbeg=ev.blkend=lineno(y)+1; /* pseudoblock */
        ev.blkend++;
        y1 = y+1;
        while (ev.blkend < ev.nolines)    { /* look for para */
            if (strncmp(curr(0, y1++), "        ", TAB) == 0)
                break;
            ev.blkend++;
        }
        --ev.blkend;
    }
    if (lineno(y) != ev.blkbeg-1)
        x = 0;
    x1 = x;
    cp1 = cp2 = ev.topptr + (ev.blkbeg - 1) * ev.wwd + x;
    cpend = ev.topptr + ev.blkend * ev.wwd;
    while (cp2 < cpend)    {
        while (*cp2 == ' ' && cp2 < cpend)    {
            if (firstline)
                *cp1++ = *cp2, x1++;
            cp2++;
        }
        firstline = FALSE;
        if (cp2 == cpend)
            break;
        /* ---- at a word ---- */
        while (*cp2 != ' ' && cp2 < cpend)    {
            if (x1 >= ev.wwd-1)    {
                /* wrap the word */
                svcp = cp1 + (ev.wwd - x1);
                while (*--cp1 != ' ')
                    *cp1 = ' ',    --cp2;
                x1 = 0;
                ev.blkbeg++;
                cp1 = svcp;
                if (y < ev.wdo->ht)
                    disp_line(y++);
            }
            *cp1++ = *cp2++;
            x1++;
        }
        if (cp2 < cpend)
            *cp1++ = ' ', x1++;
    }
    while (cp1 < cpend)
        *cp1++ = ' ';
     ev.blkbeg++;
    if (y < ev.wdo->ht)
        disp_line(y++);
    firstline = ev.blkbeg;
     if (ev.blkbeg <= ev.blkend)    {
        delete_block();
        display_text(y);
    }
    ev.blkbeg = ev.blkend = 0;
    if (firstline)
        display_text(0);
}

/* ------- test for a blank line ---------- */
static int blankline(int line)
{
    char *cp = ev.topptr + (line-1) * ev.wwd;
    int x = ev.wwd;
    while (x--)
        if (*cp++ != ' ')
            break;
    return !(x > -1);
}

/* ------------- delete a word -------------- */
static void delete_word()
{
    int wct = 0;
    char *cp1, *cp2;

    cp1 = cp2 = curr(ev.curr_x, ev.curr_y);
    if (*cp2 == ' ')
        while (*cp2 == ' ' && ev.curr_x + wct < ev.wwd)
            wct++, cp2++;
    else    {
        while (*cp2 != ' ' && ev.curr_x + wct < ev.wwd)
            wct++, cp2++;
        while (*cp2 == ' ' && ev.curr_x + wct < ev.wwd)
            wct++, cp2++;
    }
    movmem(cp2, cp1, ev.wwd - ev.curr_x - wct);
    setmem(cp1 + ev.wwd - ev.curr_x - wct, wct, ' ');
    disp_line(ev.curr_y);
}

/* ---------- delete a line --------------- */
static void delete_line()
{
    char *cp1, *cp2;
    int len;

    cp1 = ev.bfptr + ev.curr_y * ev.wwd;
    cp2 = cp1 + ev.wwd;
    if (cp1 < ev.lstptr)    {
        len = ev.endptr - cp2;
        movmem(cp2, cp1, len);
        ev.lstptr -= ev.wwd;
        setmem(ev.endptr - ev.wwd, ev.wwd, ' ');
        display_text(ev.curr_y);
    }
}

 /* ---------- delete a block ------------- */
 static void delete_block()
 {
     char *cp1, *cp2;
     int len;

     if (!ev.blkbeg || !ev.blkend)    {
         error_message("No block marked ...");
         return;
     }
     cp1 = ev.topptr + ev.blkend * ev.wwd;
     cp2 = ev.topptr + (ev.blkbeg - 1) * ev.wwd;
     len = ev.endptr - cp1;
     movmem(cp1, cp2, len);
     setmem(cp2 + len, ev.endptr - (cp2 + len), ' ');
     ev.blkbeg = ev.blkend = 0;
     ev.lstptr -= cp1 - cp2;
 }

 /* ------- move and copy text blocks -------- */
 static void mvblock(int moving)
 {
     char *cp1, *cp2, *hd;
     unsigned len;

     if (!ev.blkbeg || !ev.blkend)    {
         error_message("No block marked ...");
         return;
     }
     if (lineno(ev.curr_y) >= ev.blkbeg-1
             && lineno(ev.curr_y) <= ev.blkend-1)    {
         error_message("Don't move/copy a block into itself");
         return;
     }
     len = (ev.blkend - ev.blkbeg + 1) * ev.wwd;
     if ((hd = malloc(len)) == NULL)
         return;
     cp1 = ev.topptr + (ev.blkbeg-1) * ev.wwd;
     movmem(cp1, hd, len);
     cp2 = ev.topptr + lineno(ev.curr_y) * ev.wwd;
     if (moving)    {
         if (lineno(ev.curr_y) > ev.blkbeg-1)
             cp2 -= len;
         delete_block();
     }
     if (cp2+len <= ev.endptr)    {
         movmem(cp2, cp2 + len, ev.endptr - cp2 - len);
         movmem(hd, cp2, len);
         ev.lstptr += cp1 - cp2;
     }
     else
         error_message("Not enough room...");
     free(hd);

    ev.blkbeg = ev.blkend = 0;
    display_text(0);
}

/* ------- find the last character in the buffer -------- */
static void findlast()
{
    char *lp = ev.endptr - 1, *tp = ev.topptr;
    while (lp > tp && *lp == ' ')
        --lp;
    if (*lp != ' ')
        lp++;
    ev.lstptr = lp;
}

/* -------- carriage return -------- */
static void carrtn(int insert)
{
    int insct;
    char *cp = curr(ev.curr_x, ev.curr_y);
    char *nl = cp+((cp-ev.topptr)%ev.wwd);
    int ctl = 2;
    if (lineno(ev.curr_y) + 2 < ev.nolines)
        if (insert && nl < ev.endptr)    {
            insct = ev.wwd - ev.curr_x;
            while (ctl--)    {
                if (ev.endptr > cp + insct)    {
                    movmem(cp, cp+insct, ev.endptr-insct-cp);
                    setmem(cp, insct, ' ');
                }
                else if (ctl == 1)
                    setmem(cp, ev.endptr - cp, ' ');
```

```
                cp += insct * 2;
                insct = ev.curr_x;
            }
        }
    ev.curr_x = 0;
    downward();
    if (insert)    {
        ev.text_changed = TRUE;
        test_para(0);
        display_text(ev.curr_y-1);
        if (lineno(ev.curr_y) + 2 < ev.nolines)
            if ((ev.lstptr + ev.wwd) <= ev.endptr)
                if (ev.lstptr > curr(ev.curr_x, ev.curr_y))
                    ev.lstptr += ev.wwd;
    }
}

/* ------- move the buffer offset back one position ------ */
static void backspace()
{
    if (ev.curr_x == 0)    {
        if (ev.curr_y)
            ev.curr_x = ev.wwd - 1;
        upward();
    }
    else
        --ev.curr_x;
}
/* -------- move the buffer offset forward one word ------ */
static void fore_word()
{
    while (*ev.nowptr != ' ')    {
        if (spaceup() == 0)
            return;
        if (ev.curr_x == 0)
            break;
    }
    while (*ev.nowptr == ' ')
        if (spaceup() == 0)
            return;
}

static int spaceup()
{
    if (ev.nowptr >= ev.lstptr)
        return 0;
    ev.nowptr++;
    forward();
    return 1;
}

/* ------- move the buffer offset backward one word ------ */
static void back_word()
{
    spacedn();
    while (*ev.nowptr == ' ')
        if (spacedn() == 0)
            return;
    while (*ev.nowptr != ' ')    {
        if (ev.curr_x == 0)
            return;
        if (spacedn() == 0)
            return;
    }
    spaceup();
}
static int spacedn()
{
    if (ev.nowptr == ev.topptr)
        return 0;
    --ev.nowptr;
    backspace();
    return 1;
}

/* ----- move the buffer offset forward one position ----- */
static void forward()
{
    int ww = ev.wwd;
    if (++ev.curr_x == ww)    {
        downward();
        ev.curr_x = 0;
    }
}

/* ------- move the buffer offset down one position ------ */
static int downward()
{
    if (ev.curr_y < ev.wdo->ht - 1)    {
        ev.curr_y++;
        return 1;
    }
    else if ((ev.bfptr + ev.wsz) < ev.endptr)    {
        ev.bfptr += ev.wwd;
        if (do_display_text)    {
            scroll_window(1);
            disp_line(ev.wdo->ht-1);
        }
        return 1;
    }
    return 0;
}
/* -------- move the buffer offset up one position ------ */
static void upward()
{
    if (ev.curr_y)
        --ev.curr_y;
    else if ((ev.topptr + ev.wwd) <= ev.bfptr)    {
        ev.bfptr -= ev.wwd;
        if (do_display_text)    {
            scroll_window(0);
            disp_line(0);
        }
    }
}
```

```
/* ---- display lines in a window ------ */
static void display_text(y)
{
    while (y < ev.wdo->ht)
        disp_line(y++);
}

/* ---------- Display a line -------- */
static void disp_line(int y)
{
    char ln[81];

    if (lineno(y) >= ev.blkbeg-1)
        if (lineno(y) <= ev.blkend-1)    {
            textcolor(BLOCKFG);
            textbackground(BLOCKBG);
        }
    movmem(ev.bfptr+y*ev.wwd, ln, ev.wwd);
    ln[ev.wwd] = '\0';
    writeline(2, y+2, ln);
    textcolor(TEXTFG);
    textbackground(TEXTBG);
}
```

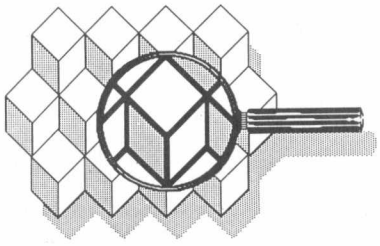
End Listing Two

Listing Three

```
/* --------- testedit.c ------------ */
#include <stdio.h>
#include <mem.h>
#include <conio.h>
#include "window.h"
#include "editor.h"

#define LNS 40                  /* number of editor lines   */
#define WD  60                  /* length of an editor line */
#define LF  (1+(80-WD)/2)       /* leftmost column          */
#define TP  (1+(25-LNS/2)/2)    /* top row                  */
#define RT LF+WD+1              /* rightmost column         */
#define BT TP+LNS/2+1           /* bottom row               */

char notes[LNS*WD];
void main(int, char **);

void main(int argc, char **argv)
{
    FILE *fd;
    if (argc > 1)    {
        setmem(notes, sizeof notes, ' ');
        if ((fd = fopen(argv[1], "r")) != NULL) {
            fread(notes, WD, LNS, fd);
            fclose(fd);
        }
        clear_screen();
        establish_window(LF,TP,RT,BT,TEXTFG,TEXTBG,FALSE);
        if (text_editor(notes, LNS, WD) != ESC)  {
            fd = fopen(argv[1], "w");
            fwrite(notes, WD, LNS, fd);
            fclose(fd);
        }
        delete_window();
        clear_screen();
        set_cursor_type(0x0607);
    }
}
```

End Listings

OF INTEREST



Graphically Speaking

Genus has announced the release of the PCX Programmer's Toolkit, which provides documentation and software to create applications that can display, save, capture, and manipulate PCX format images.

The toolkit supports all display modes of the Hercules CGA, EGA, and VGA display adapters. It contains a set of compiler interfaces, including Microsoft C, QuickC, Turbo C, Lattice C, QuickBasic, Turbo Pascal, and Clipper. Libraries are provided that are directly linkable to the user's program, and quick libraries are provided for the Microsoft integrated compilers. The toolkit also includes over 35 routines written in assembly language capable of producing on-screen animation.

In addition to the programming libraries, several utility programs are provided to display and capture screens, cut out sections of an image, locate screen coordinates, inspect image headers, and manage image libraries. An images library manager allows images to be grouped together and be directly displayed from the library without extracting first so that necessary images may be grouped with the program.

The price of the PCX Programmer's Toolkit is $89.95, source code is an additional $100. Existing users of the PBUTILS package may upgrade for $35. Reader Service No. 20.
Genus Microprogramming
6305 Mobud Dr.
Houston, TX 77074
713-771-4914
800-227-0918

Knowledge Garden has announced that KnowledgePro, a knowledge processor that integrates HyperText and system techniques into a programming language, can now access graphic images from HyperText.

With the KnowledgePro Graphics Toolkit, an add-in product that links graphic images within KnowledgePro, a user can point at one part of a picture and see a message or another picture with further hot spots, run another knowledge base, perform calculations, execute a set of *if, then* rules, and run an external program.

The Graphics Toolkit sells for $89, and the full KnowledgePro development environment costs $495. The Graphics Toolkit requires IBM PC, XT, AT, or PS/2 systems with EGA resolution, 512K, and a hard disk. Reader Service No. 21.
Knowledge Garden Inc.
473 Malden Bridge Rd.
Nassau, NY 12123
518-766-3000

The recently released **Renaissance** Graphic Device Interface (RGDI) developer's kit centers around the TI 34010-based Rendition I Advanced Graphic Controller.

The toolkit contains TMS34010 assembler diskettes with assembly language programs; an assembler, linker, library module simulator, and user's guide; TMS34010 debugger diskette developed for Rendition I for testing 34010 code on the Rendition I board; TI Software Development Board (SDB) commands; RGDI utilities diskette with command loader to download stand-alone 34010 code that will run independently of the RGDI kernel; and control panel utilities, a tool for interactively adjusting video timing parameters.

The RGDI Developers Kits is priced at $695, and the RGDI Advanced Developers Toolkit Add-on sells for $495. Reader Service No. 22.
Renaissance GRX Inc.
Cedar Park
2265 116th Ave. NE
Bellevue, WA 98004
800-422-RENX

Graph X, Version 3.0, for the Hercules family of video cards has been released by **Hercules Computer Technology** Graph X is a collection of assembly language graphics subroutines designed to be used by programmers who wish to create monochrome graphics with a Hercules Graphics Card, Hercules Graphics Card Plus, or Hercules Network Card Plus. It also allows users to create 16-color graphics on systems equipped with a Hercules InColor Card.

Graph X subroutines can be used with programs written in interpreted Basic, QuickBasic 4.0, Turbo Basic, Microsoft C, QuickC, Turbo C, Turbo Pascal, Versions 3 or 4, Microsoft Pascal, Microsoft Fortran, Ryan-McFarland Fortran, and assembly language.

Graph X, Version 3.0, is available for $59.95; customers wishing to upgrade from earlier versions can do so for $10 but must return their original Graph X diskette or title page of the manual to Hercules. Reader Service No. 23.
Hercules Computer Technology Inc.
921 Parker St.
Berkeley, CA 94710
415-540-6000

Soft Evolutions has released HGC Tools, a toolkit that provides more than 185 macros and functions for text, Hercules graphics, and the Hercules RamFont mode. Control of Hercules monochrome and InColor modes is supported by HGC Tools through macro assembler, C, and Pascal routines. Maintaining the operational status of all Hercules video cards within and between programs (including TSRs) is possible using the HGC Tools' Hercules Status Record Maintenance System. The product costs $79.95. Reader Service No. 24.
Soft Evolutions Inc.
484 Lake Park Ave. #17
Oakland, CA 94610
415-523-8247

Power Tools

Rational Systems has released DOS/16M, Version 2.0, which enables developers working under MS-DOS to create programs that directly address the 16 Mbytes of memory available on 80286 and 80386 PCs.

Rational claims that the DOS/16M, Version 2.0, boasts the reduction of DOS memory overhead to 24K Bytes for tasks running under DOS/16M; more terminate-and-stay-resident (TSR) support allowing TSRs of any size to be moved into extended memory and occupy 24K Bytes of DOS memory; support for several additional compilers; support for all currently marketed ATs and PS/2s (model 50 and higher) and compatibles; compatibility with most programs that use extended memory such as Desqview, AutoCad, Ramdrive, and VDisk; and enhanced debugging capabilities.

The product supports the following compilers: Microsoft C 4.0 and 5.0, Microsoft Fortran 4.0, Lattice C 3.10 and 3.20, Aztec C 4.10b, and Metaware's High C and Professional Pascal. Also supported is LinkLoc from Phar Lap Software, which allows programs with

OF INTEREST
(continued from page 132)

more than 1 Mbyte of code to work with DOS/16M.

Rational Systems has also enhanced DOS/16M's protected-mode symbolic debugger. Features that have been added include a built-in performance monitor to assist in analyzing the program's activity by counting DOS and BIOS service requests, interrupts, and CPU time for each function in a program; allowing execution history to be recorded in a disk file; and a new Files command to assist in analyzing file management problems by listing open file handles with the device type and full pathname.

Under Rational Systems' $5,000 *initial development* license, users receive replacement libraries for supported compilers, a program to convert an .EXE file into an .EXE executable file under protected-mode, the DOS/15M symbolic debugger, and the right to distribute up to 200 copies of one converted, protected-mode program developed using DOS/16M. Source code may be licensed for $10,000. The upgrade to DOS/16M 2.0 is free to all holders of the initial development license. Reader Service No. 28.

Rational Systems Inc.
220 N Main St., 2nd Floor
P.O. Box 480
Natick, MA 01760
617-653-6006

A new interface design tool called FaceIt has been released by **Black & White International**. This product converts a complete pull-down system into a Lotus-menu style on the screen without any coding or recompiling. FaceIt takes the information from existing DBF or ASCII text files and automatically creates single windows or multi-menu systems.

FaceIt creates pop-ups, pull-downs, pop-downs, horizontal Lotus-style menus, context-sensitive help windows, and dialog boxes; creates tables from your existing data for direct viewing and data entry selection; provides each choice on the menu in a different color; automates data entry, prototyping, and design online help.

FaceIt is compatible with dBase III Plus, FoxBase+, Clipper, Quicksilver, Basic, C, Pascal, and other languages. Selling for $99, FaceIt requires an IBM PC, XT, AT, PS/2, or compatible and DOS 2.1 or later. Reader Service No. 30.

Black & White International Inc.
P.O. Box 21108
New York, NY 10129
212-787-6633

OF INTEREST

Helios Software has released Version 3.2 of its prototype/demo system called Proteus. Proteus is an integrated development environment to build demos. Proteus 3.2 is used by developers for both design prototypes and marketing demos.

The product offers video effects such as wipes, fades, splits, sound, simulated input, optional delays, and support of Oakland's C-scape. It also includes on-screen help, system support, documentation, and text drawing commands. Users can type all 256 characters directly from the keyboard and draw line patterns with the cursor.

Proteus 3.2 also allows users to define the control logic and test their demos in its debugger. Other features are a runtime version, a screen grabber, a keyboard enhancer, source code, and conversion utilities.

Selling for $99, Proteus 3.2 requires an IBM PC or compatible, color or monochrome display adapter, 384K (256K runtime version), and DOS 2.1 or later. Reader Service No. 29.

Helios
P.O. Box 22869
Seattle, WA 98122
206-324-7208
800-634-9986

Intel Corporation has added Iceview to its I²Ice in-circuit emulator. Iceview, a graphical user interface featuring pull-down menus, display windows, and command line prompts, is now included with all standard I²Ice kits.

The Intel Integrated Instrumentation and in-circuit emulation (I²ICE) system supports real-time testing and debugging of systems that use the 8086, 8088, 80186, 80188, and 80286 microprocessors at speeds up to 10 MHz. Among its standard features are source-level symbolic software debugging, software and hardware breakpoints, execution trace, and bus and data trace.

The combination of the two products allows the user to open and maintain windows for immediate access to application source display, execution trace, registers, and other debugging information. Users can view data as it changes at breakpoints or by stepping through their programs.

Iceview is available to registered I²ICE customers as an upgrade through December for $495. Volume pricing is available. Reader Service No. 32.

Intel Corp., Literature Dept., W469
3065 Bowers Ave.
Santa Clara, CA 95051
800-548-4725

SWAINE'S FLAMES

I was up to my ears in galleys for my book on HyperTalk;
I had barely time to paradigm and no time left to squawk
(or Flame or rage) on the final page, as it is my won't to do.
So I sent a request
for a Flaming guest
and Erickson came through—again.
—Michael Swaine

What many programmers find threatening about the notion of software engineering is the George Orwellian bureaucracy that seems to go with the territory. The inevitable mediocrity of management by committee, decision paralysis, and creative straightjacketing are the most common concerns voiced by lone programmers who find themselves up against the backside of software engineering.

But, you know, there's really no inherent cause/effect relationship between software engineering and a bloated bureaucracy. Software engineering is nothing more than the application of some proven engineering principles (or project management methodologies) to the development of software. The goal (a worthy one, I might add) is to maintain control of an often massive and potentially chaotic project. If anything, software engineering *should* enable more programmers to be creative. Indeed, mundane tasks could be turned over to those tireless computers.

Earlier this year, we considered the muddy waters surrounding software engineering worthy of examination on two fronts: by addressing some specific SE topics in our regular September issue and by devoting a more indepth look at SE-related questions in a special issue, entitled *The Dr. Dobb's Software Engineering Sourcebook.*

As it turns out, getting the joys of the software engineering message across is the goal of the Software Engineering Institute, a DoD (Department of Defense, to you civilian types) sponsored organization at Carnegie-Mellon University. Its mandate—to explain and promote the notion of software engineering as a good thing. What better place, or so I thought, to begin our examination of the topic.

Inasmuch as I've heard SEI speakers at technical conferences (and they've usually given informative presentations), it seemed to me that a four page or so article shouldn't be all that difficult for someone at the SEI to write. A quick look at the proceedings of one such conference produced a likely candidate on the SEI staff and, after several calls, I got him on the phone. At his suggestion, I called the SEI press liaison (PL) officer who was delighted to hear from me, especially since the SEI had recently decided to make itself more widely known to the computer world outside the DoD.

From here the skein gets a bit tangled.

The PL gave me the name of another person at SEI who would be likely to either write the article or (at the very least) could suggest someone who could. I called person Number 3 and, yes indeed, it seemed like a smashing idea, and he'd sure like to participate. . . . But alas, no, he didn't have the time to write the article. But, oh yes, he just happened to have the names of two or three other folks. He'd let them know and someone would get back to me.

After several days, one of these fourth-tier referrals called and bit the bullet. Yeah, you bet, he'd love to write the article. No problem in meeting such a generous deadline.

Time passed. Every couple of weeks for a couple of months, I diligently phoned the writer (contact Number 4, if you're keeping track) to find out the status of the article. No problem, trust me, says he. Everything's moving along just fine. At least that's what he said.

Then came a period of darkness. For a couple of weeks no one answered his phone and my messages weren't returned. The smoking lamp was not lit. I was becoming more than a bit concerned since the deadline was marching steadily forward. You see, I was counting on the SEI article for our September issue. Finally, much to my surprise, he called. The article was finished, but it was caught up in the SEI's internal editing process. He had no idea how long that would take. This was not a good sign!

After a few days, I located the editor who had the article. Yes he had it, no, it wasn't high on his list of priorities. By this time we'd already missed the deadline for the September issue. But I was still shooting for the *SE Sourcebook.*

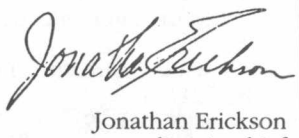
More time passed. Still no word. So I called back the press liaison who put me on this train going nowhere. After a couple of days he called back to say that the editor had been reprioritized in my favor, and that the editor's supervisor was now involved in the project.

Even more time passed. After another couple of days, I got a call from an editor I'd never talked to before (don't worry, I've lost track too) who had apparently taken over from the prior editor. The new editor said that he had called several meetings to discuss the article and that a team of editors had been assigned to work on it. Once they had finished, the article then would be sent to the technical team who would review it for *technical* accuracy. Then it would be sent back to the editing team who would incorporate the comments and reedit the article. Of course, meetings with the original author would then be necessary to ensure that the various committees had not altered the author's original meanings.

Now, it doesn't bother me that an individual writer misses a deadline. Anyone involved with publishing is used to that and, as folks around here know, I've missed a few myself. One thing that did bother me was that no one in the SEI bureaucracy seemed willing or capable of making a decision. Nor did anyone seem to have the authority to unjam the logjam.

The ultimate question concerns whether or not any conclusions can be drawn from all of this. In a purely personal sense, my worst suspicions about software engineering have been, to some degree, confirmed. Yes, software engineering apparently does spawn bureaucracies that create many of the same sort of problems the methodologies are intended to prevent.

In a more pointed sense, the SEI seems caught up in a self-defeating process whereby they can't dispel the myths about software engineering because they are too busy living the myth. Too bad. By the way, I'd still like to see the article—if it ever gets out of committee, that is.



Jonathan Erickson
editor-in-chief

#146 DECEMBER 1988
2.95 (3.95 CANADA)

ANNUAL OPERATING SYSTEMS ISSUE

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER

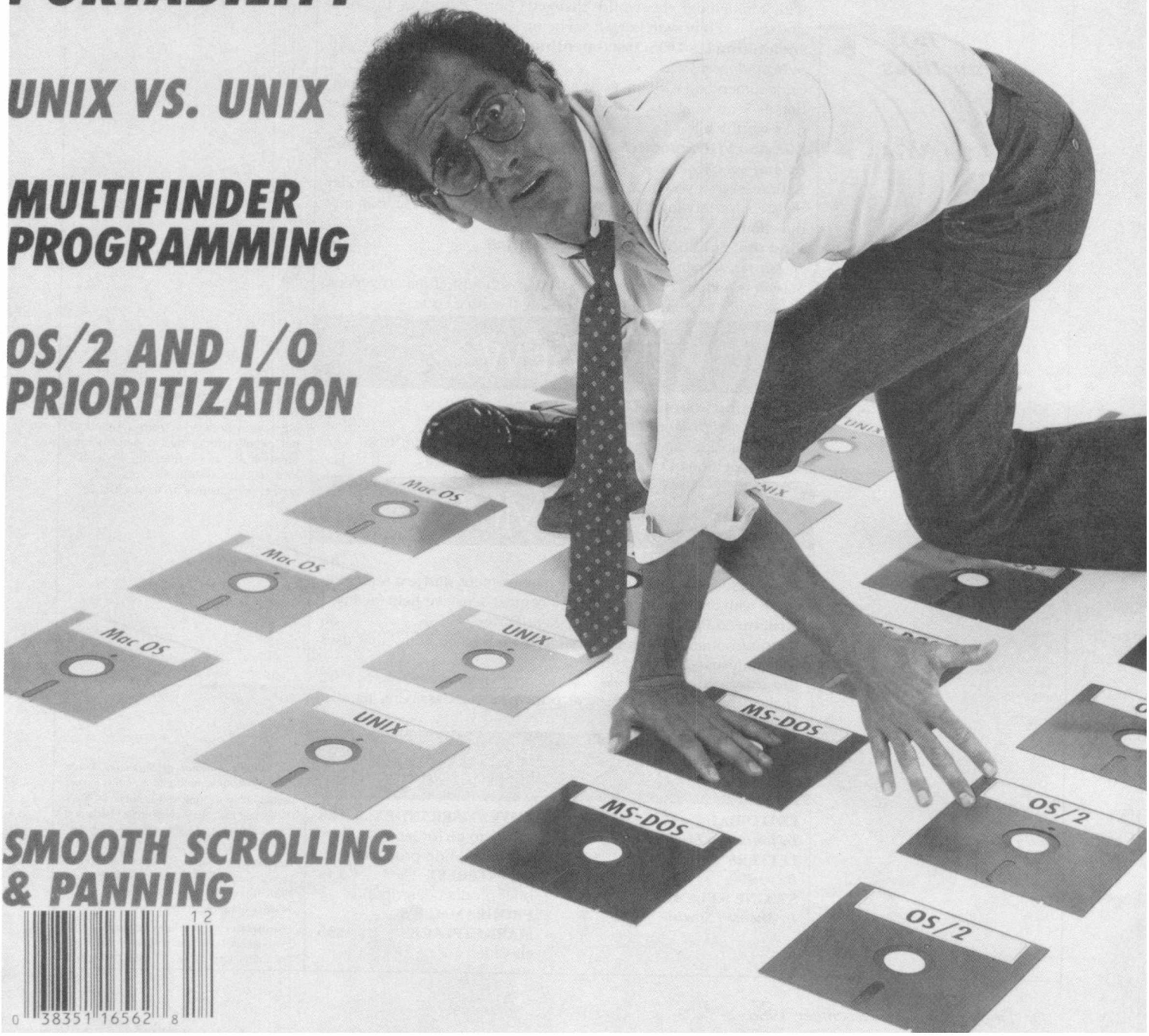
PROGRAMMING FOR PORTABILITY

UNIX VS. UNIX

MULTIFINDER PROGRAMMING

OS/2 AND I/O PRIORITIZATION

SMOOTH SCROLLING & PANNING



12
0 38351 16562 8

ARTICLES

Portability ▶ **Building Software for Portability** **18**
by Greg Blackham
Greg examines three critical issues related to software portability: the operating system, the hardware, and the compiler.

Unix ▶ **Unix vs. Unix** **28**
by Donnalyn Frey
The confusion over Unix system standards doesn't mean you have to put off writing software until the dust settles. Donnalyn presents a strategy that prevents you from getting left behind when/if a standard eventually emerges.

OS/2 ▶ **Writing OS/2 Applications with I/O Privileges** **36**
by Ray Duncan
OS/2's built-in I/O drivers may not provide all the power you need. Ray shows how to bypass OS/2 to take direct control of peripheral devices.

MultiFinder ▶ **Writing Programs for MultiFinder** **46**
by Chris Derossi
With MultiFinder the world changed; Chris describes the new landscape and how to keep from being led astray.

DOS Functions ▶ **Spelunking MS-DOS: Documenting the Undocumented** **56**
by Scott Robert Ladd
Undocumented DOS functions aren't exactly for the faint of heart. But, as Scott suggests, there are times when the payoff makes the risks worthwhile.

EGA/VGA ▶ **EGA and VGA Smooth Scrolling and Panning** **62**
by Andrew Chalk
Andrew shows you how to manipulate EGA/VGA video controller registers for moving the viewing window a pixel at a time in any direction.

Find that Function —From Inside Brief! **70**
by Marvin Hymowech
Marvin is with us once again, this time with a Brief macro version of his spiffy function finder for revising C source code.

REVIEW

C_Talk ▶ **Examining Room: A Class-ier C** **74**
coordinated by Ron Copeland
Ernie Tello looks at the latest version of C_Talk from CNS —an implementation of C in a Smalltalk-like environment.

COLUMNS

C Programming *by Al Stevens* **81**
Al discusses menu structuring, file management, and text searching along with an implementation of a context-sensitive help facility.

Structured Programming *by Kent Porter* **86**
Kent introduces Xref, a symbolic cross-reference utility that uses some unusual data structures.

Programming Paradigms *by Michael Swaine* **92**
Michael presents some general principles for classifying parallel processing algorithms.

FORUM

EDITORIAL **6**
by Jonathan Erickson

LETTERS **12**
by you

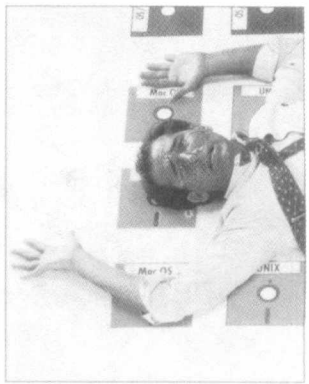
SWAINE'S FLAMES **144**
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX **128**
where to go for more information on products

OF INTEREST **134**
brief product descriptions

PROGRAMMER'S MARKETPLACE **136**
classified ads



Next Issue

We'll kick off the new year by examining neural networks from a practical perspective, focusing on pattern recognition and noise filtering. We'll also look at Unix system Streams and see how C++ measures up to Modula-2.

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly, except semimonthly in June and October by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Postmaster: Send address changes (form 3579) to *Dr. Dobb's Journal*, P.O. Box 3771, Escondido, CA 92025. **ISSN 0888-3076**

EDITORIAL

By now, you've probably read more about Steve Jobs' NeXT computer than you ever wanted to read. So far the computer (and Jobs) have been on the cover of several magazines. At least two books about it are in the works, and I've heard that at least three more book publishers are pursuing NeXT titles. Much of the initial coverage of the machine has been positive because most folks aren't quite sure yet about the nature of the beast. As programmers begin working with the machine (and reveal their findings), we should start seeing more in-depth analysis of the computer.

Until then, the significance of the NeXT computer, particularly when compared to other computers introduced recently, is that the NeXT system seems to deserve many of the accolades that have been heaped upon it. The NeXT is a fascinating computer and, even if it ultimately fails in the marketplace (and I have no reason to expect that it will), it is an important milestone because NeXT significantly ups the stakes in the computer game.

Even though it's already been argued that, technologically speaking, the computer is less innovative than NeXT would like us to believe (after all, digital signal processing chips have been around a long time, and the NeXT computer doesn't even use the most recent version of the Motorola chip —not to mention, as Jobs readily admits, the architecture itself is based on existing mainframe design principles), the NeXT computer is still more than an incremental step forward. Because of this, computer manufacturers will have to react to it in one way or another over the next couple of years or face being left behind over the next decade.

While it's the hardware of the system that first catches your attention —the 256-Mbyte erasable read/write optical storage drive, the "megapixel" display, the two VLSI chips that provide a mainframe-on-a-chip, the built-in digital signal processing chip, the 68030 CPU, the 400 DPI laser printer, and so on — it is still the software that will determine end-user acceptance (or dismissal) of the computer. And therein lies our interest.

For starters, Jobs and his cohorts built the system around Display Postscript and Unix, specifically the Mach version of the operating system. Next, they provided an easy-to-develop and easy-to-use object-oriented software environment called NeXTStep (written in Stepstone's Objective-C) that consists of a window server (which manages all on-screen image drawing, using Display Postscript), a workspace manager (which provides a graphical user interface [UI], using menus and icons to hide the complexity of Unix), an application kit (with pre-defined and user-defined objects to construct UIs), and an interface builder (to interactively create UIs). As Jobs correctly points out, most of the developers' efforts currently deal with the user interface. Jobs claims that the NeXTStep environment will cut this down to next to nothing so that programmers can devote their time to developing innovative applications instead of fighting UIs.

In one overly simplistic comparison, Jobs has compared the 18 or so NeXTStep objects to the over 400 Macintosh toolbox routines. His point is that it is easier for developers to create applications if they have fewer functions to worry about. It seemed, however, that his distinction wasn't quite on the mark because what NeXTStep seems aimed at is "end-user programming" not "professional software development." This isn't to say that end-user programming isn't important to the success of the machine (the argument could be made that end-user programming is what computing is all about anyway), but what Jobs didn't talk about was who will create the toolkits for end-user programmers and how those toolkits will be developed.

Putting the technical mysteries of the machine aside, there are all kinds of unanswered questions, particularly for software developers. For one thing, software developers must decide whether or not the higher education market (the avowed market for the NeXT computer) is big enough to support a specialized computer system. Jobs insists that it is, calling higher education a Fortune 500 company under another name, and he is betting his millions on that premise. However, to the probable relief of third-party developers, Jobs is hedging his bets because he has licensed the software (application-development tools

Editorial
Editor-in-Chief *Jonathan Erickson*
Managing Editor *Monica E. Berg*
Senior Technical Editor *Kent Porter*
Senior Editor *Ron Copeland*
Editorial Assistant *Kathleen Evans Ralston*
Technical Editor *Richard Relph*
Contributing Editors *Al Stevens*
Martin Tracy
David Betz
Tom Genereaux
Copy Editors *Rhoda Simmons*
Kevin Shafer
Editor-at-Large *Michael Swaine*

Art/Production
Director *Larry L. Clay*
Art Director *Michael Hollister*
Assoc. Art Director *Lisa Schneider*
Technical Illustrator *Lynn Sanford*
Typographers *Lorraine Buckland*
Margaret Anderson
Cover Photographer *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Direct Marketing Manager- *Andrea Weingart*
Direct Marketing Coord.- *Francesca Davies*
Newsstand Manager *Sarah Frisbie*
Fulfullment Coord. *Joan Raspo*

Administration
Vice President of Finance *Kate Deschamps*
Credit Manager *Betty Arsene*
Controller *Mary Collopy*
Accounting Supv. *Renata Kernke*
Accounts Receivable *Wendy Ho*
Accounts Payable *LuAnn Rocklewitz*

Marketing/Advertising
Director *Ferris Ferdon*
Advertising Coordinator *Patricia Albert*
Marketing Assistant *Sara Noah Ruddy*
Account Managers *see page 128*

Publisher
Peter Hutchinson

Article Submissions: Send manuscripts and disk (with article and listings) to the editorial assistant.

***DDJ* on CompuServe:** Type GO DDJ

Subscription: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds with additional postage. Add $13 per year for surface mail to all countries or add $33 per year for airmail to Canada and Mexico; add $32 per year for airlift/surface to all other countries.

Customer Service: For subscription orders and changes of address call toll-free 800-354-8400 or write: *Dr. Dobb's Journal* subscription dept., P.O. Box 3771, Escondido, CA 92025. For all other subscriber inquiries call 800-321-3333 (in California 800-331-4164, outside the U.S. 619-485-9623). For book/software orders call 800-533-4372 (in California 800-356-2002).

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; 212-686-1520 TELEX 620430 (WUI).

Entire contents copyright 1988 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

ABC

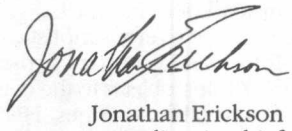
M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President *Laird Foshay*
Vice President of Publishing *William P. Howard*

EDITORIAL

and end-user interface software) to IBM, which intends on running it on its AIX Unix-based RISC and 80386 systems. Presumably, this means Jobs will sell NeXT machines to universities while IBM will sell its boxes and NeXT-compatible software to the rest of the world. All that is necessary to run software written for one system that will run on its counterpart is re-compilation.

Another confounding issue that comes to mind is the problem of software distribution. Are developers to be expected to distribute software on optical disks? That's a pretty expensive means, considering the end-user costs of blank disks will at least $50. Granted, the NeXT is a network-based workstation so most (expensive) software will be site-licensed. If there's one thing the PC revolution has shown, however, is that computer users demand the concept of one-person/one-machine, and they will want their own tools that may not be available on the network. And even if the NeXT machine had a floppy disk drive, many of the advanced applications that will be developed would require more disks than is practical. (CD-ROMs are one inexpensive possibility, as companies like Microsoft have already gleefully pointed out, and the NeXT does have those slots and SCSI port connecting peripheral devices.) Another concern is back-up procedures. One drive, one machine. How do you safely store your data? The easy answer is to back up the information onto the net, but that may not satisfy a lot of folks.

In short, NeXT has recognized that useful, yet powerful, software development tools are central to the success of any computer, and the company should be commended for providing more than just lip service to this concept. NeXT should also be commended for pushing the technology farther (or packaging it more elegantly) than many of us ever expected it would, and bringing us a little closer to what computing may be really about.



Jonathan Erickson
editor-in-chief

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION
(Act of August 12, 1970, Section 3685, Title 39, United States Code)

1. Title of publication: *Dr. Dobb's Journal.* Publication No. 08883076.
2. Date of filing: October 1, 1988
3. Frequency of issue: Monthly, except semi-monthly in June and October. (12 issues, $29.95)
4. Number of issues published annually: 14
5. Complete mailing address of known office of publication: 501 Galveston Dr., Redwood City, San Mateo County, CA 94063
6. Complete mailing address of the headquarters of general business offices of the publisher: 501 Galveston Dr., Redwood City, CA 94063
7. Names and addresses of publisher, editor, and managing editor: publisher, Peter Hutchinson, 501 Galveston Dr., Redwood City, CA 94063; editor, Jonathan Erickson, 501 Galveston Dr., Redwood City, CA 94063; managing editor, Monica E. Berg, 501 Galveston Dr., Redwood City, CA 94063.
8. Owner: M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063
9. Known bondholders, mortgages, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: Markt & Technik, Hans-Pinsel-Strasse 2, 8013 Haar bei Munich, W. Germany
10. Extent and nature of circulation:
    A. Total number of copies (net press run). Average number of copies each issue during preceding 12 months: 85,932. Actual number of copies of single issue published nearest to filing date: 93,331.
    B. Paid and/or requested circulation. 1. Sales through dealers and carriers, street vendors, and counter sales. Average number of copies each issue during preceding 12 months: 18,337. Actual number of copies of single issue published nearest to filing date: 20,574. 2. Mail subscription. Average number of copies each issue during preceding 12 months: 47,823. Actual number of copies of single issue published nearest to filing date: 50, 597.
    C. Total paid and/or requested circulation. Average number of copies each issue during preceding 12 months: 66,160. Actual number of copies of single issue published nearest to filing date: 71,171.
    D. Free distribution by mail, carrier, or other means, samples, complimentary, and other free copies. Average number of copies each issue during preceding 12 months: 2,187. Actual number of copies of single issue published nearest to filing date: 2,363.
    E. Total distribution. Average number of copies each issue during preceding 12 months: 68,347. Actual number of copies of single issue published nearest to filing date: 73,534.
    F. Copies not distributed. 1. Office use, left over, unaccounted, spoiled after printing. Average number of copies each issue during preceding 12 months: 659. Actual number of copies of single issue published nearest to filing date: 805. 2. Return from news agents. Average number of copies each issue during preceding 12 months: 16,926. Actual number of copies of single issue published nearest to filing date: 18,992.
    G. Total. Average number of copies each issue during preceding 12 months: 85,932. Actual number of copies of single issue published nearest to filing date: 93,331.

I certify that the statements made by me above are correct and complete.

Peter Hutchinson, Publisher

ARCHIVES

Ten Years Ago in *DDJ*

"No standards will be observed without discipline. To apply it, the STD bus needs a watchdog. Industry watchdogs tend to be toothless, but in this case the situation is unusual. The teeth are obvious, but the dog is not . . . commercial coercion can be simply applied by withholding use of a trademarked logo for products which do not meet established standards. Hobbyists will buy an off brand article but industrial purchasing agents will not. Proof —when did you last see a non-UL approved appliance in an office?" —*Keith Britton, "The STD Bus: A Critical Look at a Proposed New Standard,"* DDJ, *November-December 1978.*

Say What?

"Designers often fail to take into account the variety of people likely to use a computer system. A program written for other programmers (such as Unix or CP/M) can be terse, concise, and not particularly user friendly. But the average computer user who desires simply word processing capabilities or spreadsheet functions should not have to deal with obscure names for common functions." —*Terry A. Ward, "Monty Plays Scrabble,"* DDJ, *April 1985.*

Small, but Good

"The fact that the microcomputer industry has been so innovative, so creative, probably comes from the fact that the past was forgotten. The industry was new; the machines were simple and small. And simple tools and smart people produced good results. They proved that there is also a side to programming that is not represented by the Mythical Man-month —Pierluigi Zappacosta." —*Michael Swaine, "The Software Designer,"* DDJ, *October 1984.*

DR. DOBB'S JOURNAL of
COMPUTER
Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Find That Function With AWK

Dear *DDJ*,

In the article "Find That Function!" (August 1988), Marvin Hymowech provides two C programs for building and searching a file of *where-defined* information about C functions and their related source files. I don't want to denigrate what Marvin has done (because he has obviously put a substantial amount of work into the programs), but this task (among many others!) is perfectly suited to *awk*.

I have provided two *awk* programs that do basically the same thing as his C programs. Note that the code is much more compact (by an order of magnitude or two). The *bldfuncs* routine has been implemented using a different algorithm than Marvin's. I consider only those lines of C source where the last character on the line is a closing parenthesis. This set of lines includes all function definition lines. (This is coding-style dependent, but it's true for the way I and Marvin and 99 percent of others code C.) Of these lines, the ones that begin with *if*, *for*, *while*, | |, and && are discarded. This leaves just the function definition lines (with a slight possibility of a comment line).

The functionality of the second program, *getf*, has actually been enhanced in the *awk* version. One can search for function names using regular expression syntax —decidedly more powerful than wildcards. Note also that the *awk* version invokes Marvin's editor directly instead of relying on DOS environment variables.

I didn't benchmark the two versions of the program, but my guess would be that the *awk* version runs a little slower. Since *awk* is not compiled and linked, but rather interpreted, it will be inherently slower (but not slow). But this is actually an advantage when going through the code-test-debug iteration cycle.

Although *awk* is certainly not suitable for developing major applications, it is beautifully suited to developing this kind of utility program. With C, these "quick little utility functions" take on many of the aspects of major application development. With *awk*, they truly are quick and little. See Examples 1 and 2, below.

Joseph L. Kidd
San Jose, Calif.

Dear *DDJ*,

I have found that two changes should be made in Marvin Hymowech's function finding programs.

1. In *bldfuncs.c*, *get_names_one_file( )* is confused by the declaration of a pointer to a function, as in:

```
int (*point1)( )=puts;
```

It eats through the next function body and records *int* as a function. To fix this, after

```
if( c == ';' || c == ',' )/* functions
                type check declaration, */
continue;               /* so bypass it */
```

```
#-------------------------------------------------------------------------
#  Bldfuncs                Joseph L. Kidd
#  This awk program mimics the function of the same name given
#  in ''Find That Functiom!'', by Marvin Hymowech in DDJ, Aug. 1988
#
#  Usage: awk  bldfuncs.awk srcfile1 srcfile2 ...
#     where srcfile1, ... may contain wildcard characters.
#  Output: A file named funcs.txt with the format:
#     srcfile1:
#       function_1 lineno1
#       function_2 lineno2
#       ...        ...
#       function_n lineno3
#     srcfile2:
#       ...        ...
#    (where lineno1,... are the line numbers where the functions are
#      defined.)
#-------------------------------------------------------------------------

FNR == 1 { print FILENAME ":"   >"funcs.txt" }
/\)$/   {
    if ($1 ~  /^if\(*/  ||  $1 ~  /^for\(*/  ||  $1 ~  /^while\(*/ ||
        $1 ~  /^\|\|/    ||  $1 ~  /^\&\&/*)
      next
    else
      for (i=1; i<=NF; i++)
        if (x=index($i,"("))    {
          if (x==1)
            print "  "  $(i-1)  "   " FNR  >"funcs.txt"
          else
            print "  "  substr($i,1,x-1)   "  "  FNR   >"funcs.txt"
          break
        }
}
```

Example 1:

```
#-------------------------------------------------------------------------
#  GETF                        Joseph L. Kidd
#  This awk program mimics the program of the same name given
#  in the article ''Find That Function!'', by Marvin Hymowech in DDJ, Aug.
#  1988.
#  Usage: awk  getf.awk req=<req_func_name> funcs.txt
#    where <req_func_name> is the requested function name.
#    Note that <req_func_name> can be any regular expression,
#    which is significantly more powerful than DOS's wildcards.
#-------------------------------------------------------------------------
$1 ~  /:$/  { file = substr($1,1,length($1)-1)  }
$1 ~  req   { print "Pattern=\"",req, "\".   ",
                  $1, "can be found in file", file,
                  "at line", $2 "."
              req_func = $1
              req_file = file
              req_count++
}
END       { if (req_count==0)
              print "Could not find", req
            if (req_count>1)
              print "Multiple functions found."
            if (req_count==1)  {
              cmd = "b -m\"search_fwd " req_func "\" " req_file
              system (cmd)
            }
}
```

Example 2:

LETTERS

(continued from page 12)

add

```
if( c == PARENS )
/* something pointing to a function, */
continue;                /* so bypass it */
```

Actually, it could still be fooled by a function that returned a pointer to a function, as in:

```
int (*function( ))( ){ . . . }
```

or odd but perfectly legal declarations like

```
(main)(argc, argv){ . . . }
```

or

```
(main(argc, argv)){ . . . } .
```

2. In *patn_match( )*, change

```
while( *s++ )
        ;
```

to

```
while( *s )
        s++;
```

Otherwise, *s* gets incremented past the terminating zero, and the returned value can be incorrect.

I also changed *bldfuncs* to write the line number of each function definition so that *getf* could use the line number rather than the function name in the command line for the editor. I believe I'll add Unix-style pattern matching as well. Thank you for a very handy tool!

James R. Van Zandt
Nashua, New Hampshire

Dear *DDJ*,
In his article "Find That Function!", Marvin Hymowech puts his finger on a major practical problem with C, namely the poor recognizability of function definitions. I have a solution to the problem that, though so crude and low-tech it's almost embarrassing, may be of use to your readers. Namely, whenever I write or revise a C source-code file, I put the token *FUNC*, *#define*'d to be nothing either in *stdio.h* or at the head of the file at the beginning of every function definition and the comment */* FUNC */* after the *#define* in every macro definition, as in:

```
#define FUNC

FUNC char *strend(register char *s)
{
while (*s++) ;
return s;
}

#define /* FUNC */ max(a,b) ((a) >
                        (b) ? (a) : (b))
```

To find a specific definition I use one of the two *greps* as follows:

```
grep "FUNC .* strend(" *.c
```

or:

```
grep "FUNC .* max(" *.c
```

To list all the definitions I do:

```
grep "FUNC " *.c
```

To list just function definitions:

```
grep "^FUNC " *.c
```

And to list just macro defintions:

```
grep "FUNC " *.c
```

It should be fairly simple to write a program that would insert *FUNC*s into a "raw" C file, especially with *awk*. But whatever you do, always make sure that *FUNC* does not denote any real C object. If it does, and you *#define* it away, there's bound to be trouble. Normally it will show up as a syntax error, but occasionally it won't.

Margaret Armstrong
Cambridge, Mass.

Compiler Review: Clarification and Crotchet

Dear *DDJ*,
I enjoyed the compiler review in "The State-of-the-Art in Modula-2." (See September 1988.) I was encouraged to see that Kent Porter was as enthusiastic about JPI TopSpeed Modula-2 as I was. Although a 286 with hard disk and color is nicer, I found it to work quite well on a dual floppy monochrome XT system. (A small RAM disk with COMMAND.COM is quite helpful.) The multiple window environment worked well for learning the language. I could work on a program in one window, pull over a section of code into another window to test an unfamiliar feature of the language, and use a third window to look at library .DEF files.

There are a few things I felt the article may have overlooked, and so I have a request for clarification and a mention of a small "crochet."

No mention was made about Library

Source Code. JPI includes it on a third disk. Some vendors charge extra. One of the libraries is a 34-function window library. Again, some other vendors charge extra.

I had trouble with the formula 0=(C-N)/C. Running it through my calculator, I got the following results: FTL =59.03, Logitech = 62.70, TopSpeed = 83.69, Stony Brook = 88.48. As you can see, the only number that agrees with the article is the one for Stony Brook. By the way, what was the formula used for Geometric Average? I looked "Geometric Average" up in a CRC and tried to use the formula but couldn't get the results shown in the article.

One reason Logitech code sizes are so large is that all of the library object is linked in, even if only one procedure is used. If that library uses a procedure from another library, then all of the object code from the other library is also included, and so forth. (This is from page 283 of *Modula-2, A Complete Guide* by K.N. King.)

JPI TopSpeed used LONGREAL for all MATHLIB functions, which might explain the large difference in size for the FPMATH code when compared with other code sizes.

It would be nice if, in addition to the benchmarks themselves, you could show the changes needed to make the benchmarks run on each compiler. By the way, what was the environment used in making the speed and size tests? I suspect it was not a dual floppy monochrome 8088 XT compatible under DOS 3.2 at 4.77 MHz. I also suspect it was not a 386 with 1-Mbyte RAM disk and 1-Mbyte disk cache at 16 MHz. I wanted to get some idea of Modula-2 speed and code size when compared with the C compilers you had reviewed in a recent issue.

And now for the small "crochet." It would be very helpful if all compiler reviewers could pick a small suite of programs —say Sieve, Acker, and Drystone (which almost everybody uses anyway) —to run against a *standard* environment, say 286 with 20-Mbyte hard disk at 6 MHz, so that readers could get some idea of relative compiler efficiencies. That way, readers could see if the improvement of Microsoft C 6.0 (whenever it comes out) over Microsoft C 5.1 would justify the big bucks or (more interesting to me) how an efficient C compiler stacks up against an efficient Modula-2 compiler. This could be in addition to any other benchmarks they felt necessary.

Robert A. Durtschi
Jolon, Calif.

Kent responds: The algorithm used to compute geometric average is too complex to reproduce in this limited space. If anyone wants it, they can send a request and a 5 1/4-inch floppy to DDJ, *and we'll furnish the source code and .EXE for a geometric average spreadsheet that I've written in TopSpeed Modula-2.*

The test platform for the benchmarks appears at the top of page 74. We'll also furnish the code for the timing program if you request it.

The only changes made in the benchmark programs were the necessary changes in IMPORT lists, plus a couple of output statements required to adapt to TopSpeed's nonstandard calls.

As for the "crochet," I agree in principle, but there are practical problems, like what constitutes a "standard" environment. Our contributors are freelancers whose machines have little in common. We haven't the time to rerun benchmarks in the editorial offices. Thus, the best we can do in a practical sense is to make sure all benchmarks in a given article are run in the same environment.

Codeview Can Work With Turbo C

Dear *DDJ*,

Microsoft's Codeview and Turbo C 1.5 can work together. One compiles the source code with Turbo C's line numbering option on. Microsoft's linker is then used to link the object code with the /CO option enabled, which tells the linker that Codeview is going to be used on the program. The program is now ready for source level debugging.

I had purchased MASM 5.1 to help optimize some of the critical routines written in Turbo C. I ran into a bug while testing a serial port driver and was wishing for a good source level debugger that would work with Turbo C. After some tinkering, I found Codeview will work, provided one uses Microsoft's linker.

Richard J. Clark
Tetragenics Company
Butte, Montana

DDJ

Building Software for Portability

Leverage your development effort by mod-u-lar-iz-ing your code

by Greg Blackham

Software developers are finding it increasingly more important to make their applications portable across as many operating environments as possible. What developers hope for, of course, is that they will eventually be able to write only one highly portable version of software that will run on any computer. There would then be no need to endure the trauma of porting software to other operating systems and hardware.

To achieve maximum portability, software should be designed in a modular way. Code that must be changed for different environments should be isolated in separate modules. These environments should be isolated in separate modules. These environment-specific pieces can then be adapted to work properly for each environment.

Portability problems can be divided into three main categories: operating system issues, architecture issues, and compiler issues. All the issues discussed in this article fall into one or more of these categories. Furthermore, the techniques described here have been used to port WordPerfect 4.2 (which includes approximately 160,000 lines of C source code) to 12 different platforms with dissimilar operating systems and architecture.

Operating System Issues

Different operating systems provide different services to an application. This is a veritable Pandora's box of problems for which there are no easy solutions. In general, the best approach for providing portability among operating systems is to isolate all operating-system-dependent code in separate modules. You can then rewrite or adapt these modules for each environment and can maintain them separately. The vast majority of the application will generally be in modules that are not operating system dependent. For these modules, you need to maintain only one copy of source.

Every operating system has its own file system. Most file systems are hierarchical but some, such as VM/CMS on the IBM 370, are flat. Each has different, allowable file-name characters, and each allows a different number of characters in a file name. Some allow applications to assign a type to every file created. The type may mark the file as a formatted text document, a C source file, or an executable program. Developers must take care that the kinds of files an application uses are available in all desired target environments.

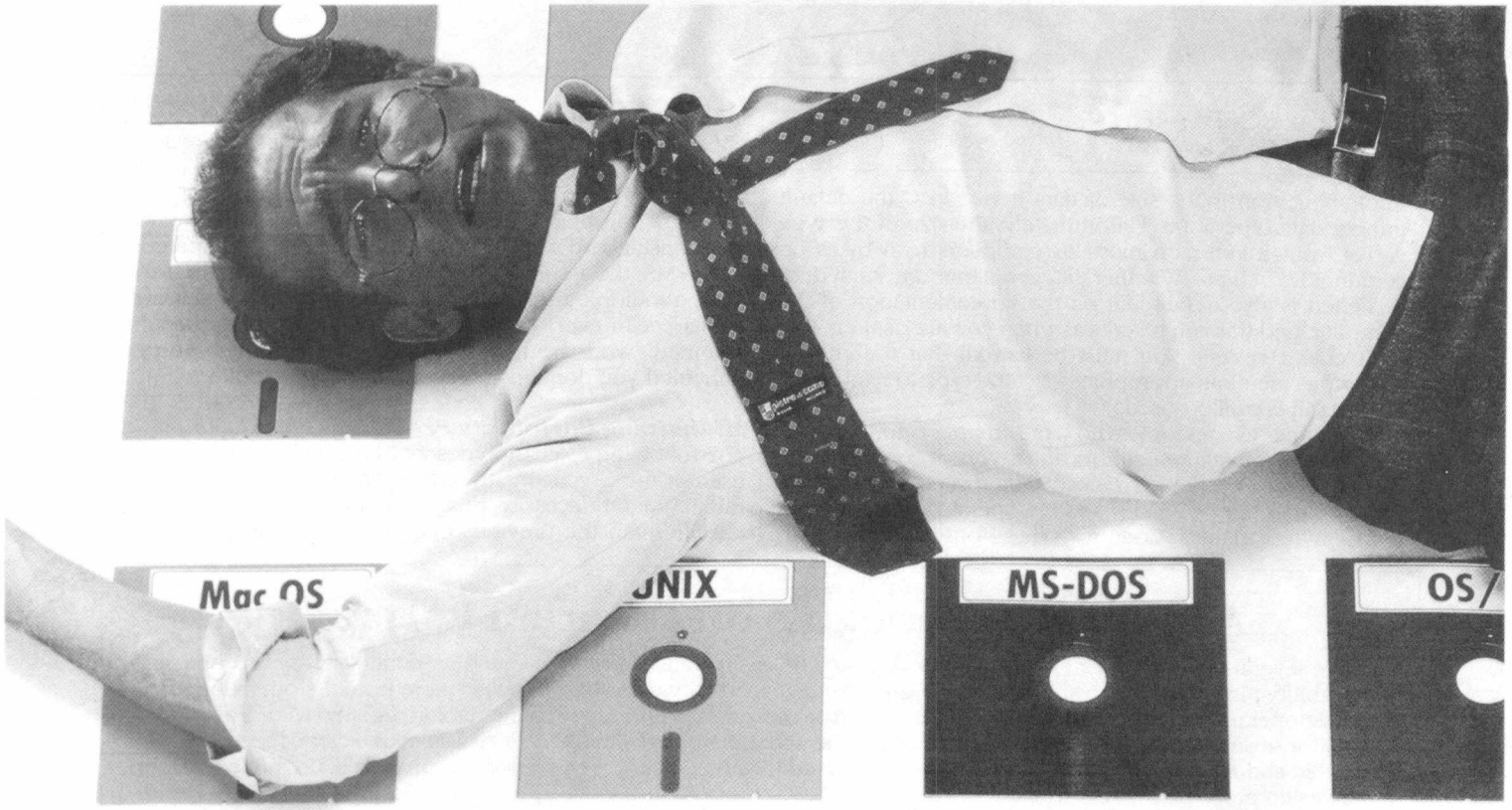
On multiuser systems file locking becomes an issue. If several users need to access one file, you must take precautions to prevent one user from overwriting the changes of another. Different operating systems provide very different levels of file locking, so to facilitate porting, you should isolate file-locking code in a separate module.

Each operating system provides a different set of system calls for getting characters from the keyboard and to the screen. Many systems provide ROM-based toolboxes or software library toolboxes such as the Unix curses package. Making screen I/O completely generic results in unacceptable performance on many systems —we have found that the best approach is to isolate screen I/O in one module and customize and optimize that module for each environment.

A related issue concerns character sets. Although ASCII is the most widely accepted standard, some environments still use other character sets, the most notable exception being IBM with EBCDIC. Even if ASCII were universally accepted, it is only a 7-bit standard. International and extended character set standards are under consideration, but the huge installed base of existing equipment supports a variety of nonstandard character sets. Even on variants of the same operating system, such as Unix, no standard character set can be counted on. Applications that make use of international, technical, or graphics characters must be written carefully to make sure they don't depend on one character set.

It is a fact of life that on some platforms an application must follow certain user interface standards if it is to be accepted by the user base. This is especially true of machines that provide a mouse-based graphical user interface. An application for the Macintosh, for example, must have a "Mac-like" user interface to be accepted. Similarly, a product for Sun workstations that does not take advantage of the mouse and windowing system is not acceptable. You must give careful consideration to user interface decisions if an

Greg Blackham is the director of Unix product development at WordPerfect Corp. He can be reached at 1555 N. Technology Wy., Orem, UT 84057.



application is to be ported to both graphical and text-based environments.

Each operating system has a distinct set of possible error conditions that can be encountered by an application. Many of these extend across all environments. For example, every operating system returns a "file not found" error when an "open" system call fails because the file does not exist. Each environment may also have its own unique set of messages. On multiuser or networking systems, an "open" might also fail because the user does not have rights to access the file, or because the file is in use by another user. Error-handling code must be flexible to handle the various exceptions appropriately.

Architecture Issues

Architecture issues arise from the instruction set available on the specific processor on which the application will run. Each processor has certain idiosyncrasies that you must consider when you write portable code.

Different chips store integer data in different ways. Most Intel chips, for example, store a 16-bit integer with the low-order byte first, followed by the high-order byte. This is commonly referred to as "little endian," because the little end comes first. In contrast, most Motorola chips store the high-order byte before the low-order byte ("big endian"). Still more variations are possible when 16-bit words are stored big endian but the bytes within the words are stored little endian. To assure portability, you should not write code that depends on the order in which integer data is stored.

Nor can you make assumptions about the size of data addresses or code addresses. It is especially dangerous to assume that data addresses are the same size as function addresses or that an address can be saved or typecast into an integer variable.

Most IBM PC compilers allow a variety of memory models to be used. This is because of the segmented architecture of the 8088 line of microprocessors. If one code segment is defined, then a function pointer will be a 16-bit item, whereas it must be a 32-bit object if multiple code segments are defined. Similarly, models are allowed that define one or multiple data segments. Nearly any combination is possible. For example, we had to change certain table-lookup utilities that were used for both code and data addresses when we ported them to an 80286-based environment.

Some architectures require that some types of integer data be stored at even byte addresses. On the Western Electric WE32100, for example, some operations will crash the application if the data is not properly aligned. Still other processors sport better performance if integer data is aligned on even byte boundaries, so compilers force alignment of data. For application programmers, this means it is unsafe to assume that two variables that are defined consecutively will be contiguous in memory.

It is often desirable, if not an absolute necessity, to share data files among environments. For example, word processing documents, spreadsheets, and database files should be stored in a format which is directly loadable on all machines where these applications run. This allows users on many different systems to share information conveniently. The architecture issues mentioned previously make this portability of data files more complicated to achieve. Byte order varies according to processor and architecture, and word alignment may cause identically defined data structures on two machines to be different sizes. Because of these possibilities, it is not wise to write data directly from internal data structures to data files. You must take care to assure that data files are written and read so that the data is correct for all architectures.

Some operating systems allow only certain types of files. Some do not allow ISAM files; others may not allow arbitrary-length files and always fill to a full physical disk block. If a file must be portable, select a file structure that is available on all systems to which it will be ported.

Compiler Issues

Several portability issues result from the various implementations of C compilers. Some of these arise from nonstandard but useful extensions, others arise from incorrect or nonstandard implementations, and still others arise in ambiguous areas of the language specification where no "right" answer is available.

The most critical portability issue arising from compiler

(continued from page 19)

implementation is the size of data items. In C the default integer data type is *int*. Unfortunately, the size of this type varies from 2 bytes on most PC compilers to 6 bytes on certain RISC chips. Whether integer items are signed or unsigned is also dependent on the implementation of the compiler, and this can be critical when you are dealing with 8-bit character sets. You must be careful that the chosen data types are of an appropriate size and type to handle the data for all compilers used.

Sometimes even people who write compilers foul up, and certain implementations simply don't work according to the language specification. On most 80386-based C compilers, for example, the *=, /=, |=, and &= operators do not work with 1-byte variables. Code with these operators compiles to invalid assembly code. In cases like this, you can choose either to wait for "the fix," which may delay the release of your product, or you can work around the compiler problem by using a different construct.

Another problem occurs when a compiler is implemented in a nonstandard way. As an example, in the C compiler for VAX/VMS, the assumption is made that all data is shareable among multiple users unless the data is explicitly declared unshared. In every other C compiler, the exact opposite is assumed. Work-arounds for this kind of problem can be difficult if you don't maintain separate source code.

Minimizing Portability Problems

The following discussion describes the techniques we have used in the C versions to reduce portability problems to a minimum. Of necessity, this discussion is more C language specific than the previous one; however, analogous tech-

A Few More Thoughts on Program Portability

Here are several more broad categories of portability problems that an application developer may choose to consider. Usually these decisions revolve around issues that should be resolved as early in the design process as practical, such as which machines and devices will the application work with, what countries will the application sell into, how sensitive will the customers be to application performance, and what other products will the application have to work with.

Device Independence

The subject of portable graphics applications is broad and difficult to handle in a small space. There are several application environment products, however, that make it easier to write graphical applications that can run with a variety of different devices, ranging from simple monochrome screens to higher resolution color devices. One such environment is Digital Research's Graphical Environment Manager (GEM), which runs on a diverse range of target systems, including Motorola-based Atari's, as well as Intel-based IBM PCs and clones.

Internationalization

As the article mentioned, appropriate error analysis and reporting is a key to writing professional applications. If you plan to sell the program into markets where your users don't speak English, you need to make sure that your application handles foreign languages. One consideration mentioned was the ability to handles alternate character sets. You should be aware that Japanese and Far Eastern languages use 16-bit character sets, which means your application should be careful not to assume that all input and output will be limited to 7- or 8-bit characters.

A more important consideration is that the program's messages (including errors) should be easily translated into foreign languages. This works best when all error messages are grouped in a single file and are NULL delimited rather than fixed length.

Performance-Sensitive Optimizations

In order to compete effectively in the marketplace, your program must perform well. Modularity plays an important part here, too. Those portions of a product that are called extremely often should be isolated in their own modules. A version of the module can be kept in a higher-level language like C. However, you may want to allow for time in the development of your application to hand-code the performance-sensitive modules into assembly code. Thus, you will still be able to quickly port the product to the next environment by using the high-level language module, while still getting the performance you need in more competitive environments by using the assembler version. Also, by carefully designing and analyzing your program, you will probably find a relatively small amount of code requiring hand-tuning. Many ports of the Unix operating system, for example, perform well with less than five to ten percent of the operating system hand-coded in assembler.

Adhering to Standards

No man is an island and neither is your program (especially in these days of open systems). Programs that use standard operating system calls or data formats wherever possible will port more easily and work well with other programs in a variety of environments.

The ANSI C standard run-time library defines a very useful set of I/O interfaces that can be implemented on a wide variety of platforms. Your application should use the standard calls as much as possible. If your application needs more from the operating system interface than is provided by ANSI C, POSIX extends beyond the ANSI C functions to provide a Unix-like operating system interface that will be ported to a number of Unix and non-Unix systems.

For data independence, Unix defines a model of standard input and output and provides the ability to send the standard output of one application into the standard input of another. Using a standard character set like ASCII rather than a binary data representation facilitates this kind of interaction on Unix and non-Unix systems alike.

There are a variety of other standards that will become increasingly important across a variety of different environments. To represent documents, your application may need to handle the standard generalized markup language (SGML) format, which is specified by the American Association of Publishers. To exchange graphical vector information, you may want to use the graphical metafile format, used by GEM applications, or the computer graphics metafile (CGM) standard format. To handle image and scanned data, you will probably want to handle tagged image file format (TIFF) files.

One final note. Writing portable programs is hard and can be very expensive. Knowing how to write portable programs will take you most of the distance, and using application environments and tools like C and GEM will get you even farther. Knowing how much effort to expend to write portably, though, is up to you. —B.F.

Bill Fitler is a senior staff engineer and heads the X/GEM application team at Digital Research Inc. He can be reached at 70 Garden Ct., Box DRI, Monterey, CA 93942.

(continued from page 20)

niques are possible in nearly any high-level programming language.

The *typedef* operator available in C allows us to create our own data type names so that, when a data item is defined, we know its exact size. Example 1, this page, shows an excerpt from the file machine .h, which contains our machine-dependent data-type definitions. For example, a *UWORD* is an unsigned integer quantity that is 2 bytes wide, and a *BYTE* is a single-byte unsigned integer (used for 8-bit characters). All *typedef* declarations are in a separate include file that contains the appropriate types for each machine. Each time a port is made to a new machine, the data types must be defined appropriately for the new environment. This technique allows us to know the exact size of every data item defined, regardless of the environment.

```
/*  MACHINE.H                                   */
/*  MACHINE DEPENDENT DATA TYPE DEFINITIONS     */
/*  Copyright (C) 1987                          */
/*  WordPerfect Corp.                           */
/*                                              */

#ifdef  IBM-PC
    typedef unsigned char   BYTE;       /* unsigned,  8 bits */
    typedef int             WORD;       /* signed,   16 bits */
    typedef unsigned        UWORD;      /* unsigned, 16 bits */
    typedef long            DWORD;      /* signed,   32 bits */
    typedef unsigned long   UDWORD;     /* unsigned, 32 bits */
#endif


#ifdef HP
    typedef unsigned char   BYTE;       /* unsigned,  8 bits */
    typedef short           WORD;       /* signed,   16 bits */
    typedef unsigned short  UWORD;      /* unsigned, 16 bits */
    typedef int             DWORD;      /* signed,   32 bits */
    typedef unsigned int    UDWORD;     /* unsigned, 32 bits */
#endif


#ifdef  ATT6386
    typedef unsigned char   BYTE;       /* unsigned,  8 bits */
    typedef short           WORD;       /* signed,   16 bits */
    typedef unsigned short  UWORD;      /* unsigned, 16 bits */
    typedef int             DWORD;      /* signed,   32 bits */
    typedef unsigned int    UDWORD;     /* unsigned, 32 bits */
#endif

 . . .
```

Example 1: *An excerpt from the file machine.h, which provides information on the exact size of every data item defined, regardless of the environment.*

You should take great care to assure that whenever possible data files can be shared across environments. This requires that such files be read and written in a special way. As mentioned earlier, writing a data structure directly from memory to disk creates a nonportable file because the byte order in integer data will reflect the architecture and also may include "space holder" bytes to ensure word alignment.

To deal with this situation, data files can be created one byte at a time and interpreted one byte at a time when read. If the reading and writing are done in large blocks, disk access can be minimized. Example 2, page 26, illustrates how identical source code on two different processors creates radically different files. Not only does the integer data get written in a different order, but also the files are a different size because one machine aligns on 4-byte boundaries whereas the other aligns on 2-byte boundaries. Example 3, page 26, shows how you might modify the code to produce identical data files on all machines, regardless of byte order within integer data and word alignment.

To assure maximum portability, screen output and keyboard input need to be handled in a portable way. A generic library of screen and keyboard functions should be provided. These functions will produce the same result in all environments, but can be optimized to work efficiently on each platform. On machines where it is possible (such as the PC), you should write directly to screen RAM, as this method provides the best possible performance. In environ-

(continued from page 22)

ments that use 9,600-baud terminals, you can optimize a package to avoid redundant writes to the screen.

Because no standard character set exists, some internal representation should be chosen for extended/foreign characters. This allows files with these characters to be moved easily between platforms. We chose the IBM PC extended character set to ensure compatibility with our existing PC product, but any character set could be used as long as it is the same on all platforms. The screen/keyboard library handles the translation of the internal representation to the native character set.

Like screen and keyboard functions, file I/O and other operating system calls should be isolated in generic library functions. These can be customized and optimized for individual environments. This technique allows the vast majority of the source code, which does not involve operating system calls, to be environment independent.

As mentioned earlier, a variety of compiler implementations conform in varying degrees to official language specifications. Wherever possible, try to avoid all but the most standard features. In the case of C, the ANSI C standard provides functionality that was not defined in the original Kernighan & Richie specification. Not all target environments have ANSI compilers yet, so for now, avoid using ANSI extensions unless you treat them as environment-specific code and isolate them in separate modules.

Many environments provide programs that point out nonportable constructs. The most widely available tool for C is called *lint*; it is available on all Unix systems as well as with most PC C compilers. Similar tools are available for other languages and environments.

It is extremely helpful, whenever possible, to develop a portable application on several distinct platforms at the same time. Most of the problems discussed here were learned from sad experience, not from portability textbooks. If you choose several disparate platforms, most portability problems will surface early in development, which often allows you to make adjustments to the implementation before you create a lot of nonportable code.

If you can't afford the luxury of concurrent development on several platforms, try to do the most generic environment first. This varies depending on the programming language you select. In the case of C, the most generic environment is Unix. Almost all other C implementations provide Unix system call libraries. PC compilers, on the other hand, provide many semistandard constructs for segmenting programs and accessing DOS and the ROM BIOS. In my experience C code ports more easily from Unix to other environments than vice versa.

Summing the Parts

Creating applications that port easily to a variety of operating environments, though challenging, is not impossible. The techniques discussed in this article can improve porting productivity dramatically. Porting WordPerfect among Unix platforms may require only days or weeks; other ports may require only a few hours. By improving the portability of your source code, you can provide software products on more machines at a lower cost. Without portable code, many of these ports would be simply too expensive to consider.

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

```
/*********************************************************************
 *                                                                   *
 * This example program writes a data structure directly to disk.    *
 * Only 5 bytes of data are significant, but more than 5 bytes       *
 * are written to the file because of word alignment.  The           *
 * component bytes of the integer data are also written in a         *
 * different order depending on machine architecture.                *
 *********************************************************************/

#include <fcntl.h>
#include "machine.h"

/* 5 byte structure definition */
/* word alignment may occur between elements 1 and 2! */

struct {
    BYTE element1;
    UDWORD element2;
} s = {'A',             /* recognizable values for each byte */
       0x08040201};     /* each byte represented by 2 hex digits */


main()
{
    int fd;
    fd = open("data.fil",O_RDWR | O_CREAT,0666);
    write(fd,(char *)&s,sizeof(s));          /* open and write file */
    close(fd);
}

Contents of data.fil created on HP 9000 Model 350:

65,         This is the "A"
0,      This byte is inserted by the compiler for word alignment
8,      Bytes of element2 stored with the highest order byte first
4,
2,
1           Lowest order byte of element2


Contents of data.fil created on AT&T 6386:

65,         This is the "A"
0,      These 3 bytes are inserted by the compiler for word alignment
0,      The 6386 compiler aligns on 4-byte boundaries
0,
1,      Bytes of element2 stored with.the highest order byte first
2,
4,
8           Highest order byte of element2
```

Example 2: *This example program writes a data structure directly to disk.*

```
/*********************************************************************
 *                                                                   *
 * This new version of the example program writes the same data      *
 * structure to disk in a portable way.  It creates a memory image   *
 * of what the disk file should look like, then writes the file      *
 * in one block.  When writing the integer data the bytes are        *
 * written starting with the least significant byte.                 *
 *********************************************************************/

#include <fcntl.h>
#include "machine.h"

/* 5 byte structure definition */
/* word alignment may occur between elements 1 and 2! */

struct {
    BYTE element1;
    UDWORD element2;    /* 4 byte integer */
} s = {'A',             /* recognizable values for each byte */
       0x08040201};     /* each byte represented by 2 hex digits */

main()
{
    int fd,                             /* file descriptor */
        i;                              /* counter variable */
    BYTE array[5];                      /* output buffer */
    UDWORD tmp;                         /* a 4-byte integer */

    array[0] = s.element1;              /* 1-byte variables are easy */
    tmp = s.element2;                   /* Don't clobber the real data */
    for (i=1; i<= 4; i++) {             /* put each byte of the UDWORD */
        array[i] = (BYTE)(0xff & tmp);  /* in separately */
        tmp >>= 8;                      /* get next 8-bits */
    }

    fd = open("data.fil",O_RDWR | O_CREAT,0666);
    write(fd,array,5);                  /* open and write file */
    close(fd);
}


Contents of data.fil created on all machines using this code:

65,         This is the "A"
1,      Bytes of element2 stored with the lowest order byte first
2,
4,
8           Lowest order byte of element2


Note that the C compiler takes care of getting the bytes in the
correct order, no matter what the internal byte order.  The code
to read the file reverses the process with a similar "for" loop.
```

Example 3: *This version of the example program writes the same data structure to disk in a portable way.*

Unix vs. Unix

Confusion, not unity, is the byword in the Unix community as the window of opportunity stays open a little longer

by Donnalyn Frey

On May 17, 1988, the major players in the Unix community changed; AT&T was suddenly no longer the only important voice in Unix. AT&T owns the Unix operating system, but now the Open Software Foundation, the voice of several computer firms, is playing a role in the Unix community. The trouble is, though, that no one knows what the new voice is going to say or who is going to listen.

This situation started early in 1988 when AT&T, stung by its failure to penetrate the Unix workstation market, concluded an agreement with Sun Microsystems, of Mountain View, Calif. Sun is a young company with the dominant hold on the Unix workstation market. Sun agreed that AT&T could purchase up to 20 percent of Sun stock, at a premium. AT&T would have a seat on the Sun board of directors, and most important, Sun and AT&T would combine their versions of the Unix operating system. AT&T would still own Unix, but Sun and AT&T would work together on future releases. The new Unix operating system that will be released by AT&T will be created from System V, Release 3 (the latest version of the AT&T Unix operating system), SunOS (the Sun operating system), and Berkeley 4.3BSD (a version of Unix created by the Computer Systems Research Group at the University of California, Berkeley). SunOS was originally based on the 4.2BSD code, known for its networking capabilities, and has been upgraded to 4.3BSD. The new AT&T operating system will be called System V, Release 4, and is scheduled for a mid-1989 release.

Donnalyn Frey is president of Frey Communications. She is a spokesperson for the USENIX Association and can be reached at P.O. Box 2051, Fairfax, VA 22031-0051.

What the agreement between Sun and AT&T meant was that Sun, a workstation manufacturer, would be working on the new AT&T Unix operating system and thus would have access to it before any other Unix computer manufacturer. Many vendors believed that Sun would have the advantage over other manufacturers because it could write and release new applications that took advantage of the newest Unix release before anyone else could.

AT&T tried to allay these fears by meeting with the other computer and workstation manufacturers. In those meetings, AT&T stated that it would keep the manufacturers apprised of work on the new operating system so that no one would be left behind. But the companies couldn't look at the newest operating system until just before it was released, months after Sun had already worked on it. These meetings did little to allay the fears. A few companies, such as Unisys, agreed to work with Sun and AT&T. But most of the other companies were still concerned.

These companies, including Digital Equipment Corp., MIPS, Prime Computer, and several others, held meetings among themselves to discuss the situation. Their first meeting, held at the Hamilton Street location of one of the attendees, eventually led to the formation of the Hamilton Group.

The Hamilton Group

The Hamilton Group held several discussions. Many believed that the AT&T/Sun agreement threatened the future of their companies. AT&T licenses Unix to companies that produce computers that run Unix. The companies can then create their proprietary products, within some AT&T license limitations. This has led to DEC's Ultrix, Sequent Computer's Dynix, and other forms of Unix. But first the companies have to see the Unix source code before they can write their own code. The new AT&T/Sun agreement has Sun and AT&T working together on the newest version of Unix, with Sun able to write new applications months before any of its competitors.

The Hamilton Group considered several alternative strategies, including writing its own Unix, free of licensed AT&T code. There was considerable concern among the group over splitting the Unix market and creating confusion. There are already two major forms of Unix — the AT&T System V version and the University of California BDS version. The BSD version was licensed by AT&T; however, it has been split from AT&T for many years. The BSD version of Unix contains many innovations not found in AT&T Unix. System V is often preferred by businesses, with BSD preferred by research firms and universities. Xenix, the most popular version of Unix for PCs, is based on System V.

After a few months of meetings, several members of the Hamilton Group, with some other computer companies, decided to form the Open Software Foundation (OSF). Even though they were aware that this might confuse the Unix market, they believed the OSF would eventually bring a sense of order to it.

The Open Software Foundation (OSF)

The OSF was created on May 17, 1988, by seven sponsors. Each sponsor pledged $4.5 million per year over three years for the fledgling organization, and each sponsor would have a seat on

the board of directors. The original seven sponsors were IBM, Digital Equipment, Apollo Computer, Nixdorf Computer AG, Groupe Bull, Hewlett-Packard, and Siemens AG. AT&T was invited to join the organization, but it declined.

Shortly after its formation, the OSF began running full-page advertisements in the trade press announcing its existence. An OSF advertisement appearing in *MIS Week* on May 30, 1988, read: "A truly open software environment would allow business to harness the power of computers, regardless of their size or manufacturer. That's why we have formed the Open Software Foundation —to make this environment a reality." The advertisement went on to list the principles of the OSF:

- Offerings based on relevant industry standards
- Open process to actively solicit inputs and technology
- Timely, vendor-neutral decision process
- Early and equal access to specifications and continuing development
- Hardware-independent implementations
- Reasonable, stable licensing terms
- Technical innovation through university/research participation

The advertisement included an invitation to other companies to join the OSF as members, not sponsors. It concluded with the logos of each of the sponsors.

Although the advertisements announced the principles of the OSF, the Unix community was still confused and concerned. Many people wanted to know what the OSF was really about, behind all the announcements and photographs of smiling CEOs.

The federal government was especially concerned. Approximately 70 percent of all government requests for proposals (RFPs) on new data-processing systems contained a requirement for the Unix operating system. The government had been requesting System V compatibility. Would the OSF confuse government procurement of Unix systems? Would the government again be drawn into lawsuits over Unix specifications, as happened when DEC challenged a requirement for System V compatibility? The federal government specified Unix because it wanted a portable operating system for the majority of its computers. Would OSF damage this portability?

The birth of the OSF brought hundreds of questions to mind for developers, research organizations, universities, businesses, and the federal government. What did the OSF mean? What was the OSF going to do? What would AT&T do? What did it mean for users planning to purchase Unix systems? Should developers go with AT&T/Sun Unix or wait for any OSF offering? Answers to these questions were not being provided by anyone, and speculation was rampant in the trade press and in the hallways and conference rooms of countless organizations.

Many Unix observers found it interesting that DEC and IBM had joined in the founding of the OSF. DEC has been considered hostile to Unix for many years, and has always promoted its proprietary VMS operating system. The president of DEC, Ken Olsen, recently called Unix portability "snake oil," causing considerable furor in the community. Approximately 20 percent of DEC's market, however, is Ultrix (Unix). This market is growing and DEC may be responding to user demands for Unix. DEC has announced that it will retain its proprietary Ultrix system, but it will be in compliance with OSF standards. These standards, however, had not been announced at the time of DEC's statement.

IBM has similarly been considered hostile to Unix, preferring to market its own operating system. Many people wanted to know if the OSF was designed to confuse the Unix market in order to give DEC and IBM more time to promote their proprietary operating systems. The wave of the future seemed to be Unix, but if IBM didn't want that to happen, could it stop it?

Another major concern of the Unix community was IBM's reported insistence that it would join the OSF only if its Unix operating system, AIX, was used as the base of any OSF operating system offering. IBM has minimal experience in the Unix market —in fact it contracted out the initial work on AIX. There was considerable concern that any OSF product based on AIX would not be robust enough for market acceptance.

There was also concern that the OSF would never produce anything. The OSF had stated the intention of bringing its first product to market within two years. After waiting two years, the Unix community might find nothing available from the OSF or only an unusable product. A product produced by committee is often useless, especially when members of the committee have differing interests. There was considerable concern that the OSF's only product would be two years of confusion in the Unix market.

Next, the whole question of standards was raised. Because AT&T licenses Unix to many computer manufacturers and software firms, each of which may rewrite parts of the operating system and use it as the firm's proprietary Unix offering. The only requirement is that the company pay the appropriate license royalties to AT&T. Consequently, many different versions of Unix are available. Berkeley 4.3BSD is different from Sun's SunOS, which is different from Sequent's Dynix, which is different from DEC's Ultrix, and so on.

Unix users and developers have recently been working to implement standards in the Unix industry to ensure that applications written for one form of Unix will run on other forms. The Unix operating system will then be truly portable.

POSIX

The largest standards effort is the IEEE P1003.1 POSIX standard, an effort by the IEEE to create a basic standard Unix. The acronym POSIX stands for Portable Operating System Interface X, with the *X* added to make the title end in

Unix System V	Open Systems Foundation
AT&T	IBM
Sun Micro Systems	DEC
Unisys	Hewlett-Packard
Motorola	Nixdorf
Toshiba	Siemens
Prime	Hitachi
Olivetti	Phillips
Amdahl	Silicon Graphics
Control Data	Group Bull
Gould	Apollo Computer
ICL	National Semi
NCR	AMD
Fujitsu	Pacific Bell

Table 1: *Major companies endorsing Unix System V or the Open Systems Foundation*

(continued from page 29)

IX, as in Unix. The P1003.1 committee has written a standard detailing what every Unix system should include. This standard does not try to make all forms of Unix identical, merely able to run common applications among themselves, which is the goal of a portable system.

The POSIX standard has, at its core, the AT&T System V Interface Definition (SVID), a Unix standard AT&T was promoting. It also contains requirements based on the Berkeley 4.3BSD operating system, among others. The standard, several years in the making, is supported by the Unix industry, technical associations, the OSF, and the federal government and was approved by IEEE on August 22, 1988.

Systems that are POSIX compatible must pass a test suite of programs to prove their adherence to the standard. Most forms of Unix are already in compliance with most of the POSIX requirements. POSIX addresses only the basic Unix operating system, and it contains many options. It will not solve all incompatibility problems, but it will help.

The approval of POSIX helps to clarify the turbulent Unix market. AT&T, the OSF, and dozens of computer hardware and software firms support POSIX. Unix has a basic standard to follow. Someone charged with purchasing a Unix system does not have to choose between AT&T Unix or wait to see if there will be an OSF Unix. Users can specify POSIX-compliant Unix and will have a Unix system that can run applications from any number of manufacturers. The approval of the POSIX standard also frees users from reliance on a single hardware vendor. With POSIX, a system can be created with computers manufactured by several vendors.

Developers also benefit from the POSIX standard because they know what their basic Unix offerings should include for market acceptance. Developers may enhance their products within the AT&T and POSIX guidelines, but they will still be POSIX compatible.

The standard is also of importance to the federal government. The FIPS-POSIX (Federal Information Processing Standard-POSIX) standard, the government form of POSIX written by the National Bureau of Standards and Technology, was approved by the Department of Commerce on August 31, 1988. Unix suppliers have nine months to bring their systems into compliance with POSIX. Federal RFPs can now specify FIPS-POSIX for the standard Unix operating system. The federal government is assured of basically compatible Unix systems, no matter what happens in the AT&T/OSF situation. This version of FIPS-POSIX is based on the earlier POSIX standard, rejected by the IEEE P1003.1 committee. The National Institute of Standards and Technology is reviewing the POSIX standard with the intention of bringing the FIPS-POSIX standard into line with the current P1003.1 POSIX standard.

Even with the approval of POSIX, the Unix community is still unsettled. The AT&T/OSF situation continues to confuse the market. The OSF has been adding new members. Companies have been invited to join the OSF as members at a cost of $25,000 annually for profit-making corporations and $5,000 for nonprofit corporations. Many companies and a few universities have joined OSF, including Relational Technologies, Adobe Systems, Altos Computer Systems, Mitre Corporation, Stratus Computers, Toshiba America, Tecseil, Cornell University, Locus Computing, National Semiconductor, Phoenix Technologies, Concurrent Computer, Data Logic Ltd., Interactive Systems, Interfirm Graphics Systems, Mentor Graphics, the University of Southern California, 88open Consortium Ltd., Advanced Micro Devices, Booz Allen & Hamilton, Micom Interlan, Norsk Data A.S., Pacific Bell, Silicon Graphics, Stanford University, The Swedish Telecom Group, Wang Laboratories, Computer Consoles, Data General, Informix Software, and Landmark Graphics. Other companies have expressed interest in joining.

In addition, OSF has added a new sponsor to the original sponsors. The new sponsor is Hitachi Ltd. of Tokyo, Japan. The OSF had originally stated that it would accept members, but not new sponsors. However, Henry Crouse, president of OSF, believed that the strategic location of Hitachi in Japan would enhance the acceptance of OSF products in Japan.

The OSF and USENIX

The OSF itself is also joining organizations. It recently joined the USENIX Association, the technical and professional association of Unix users, at the association's semiannual technical conference in June 1988. The OSF held an information meeting at the USENIX Association conference in San Francisco to allow conference attendees to talk with the director of research, Ira Goldstein; director of development John Paul; director of strategic development, Alex Morrow. Ira Goldstein, at the time of the meeting, was the manager of research and development of Hewlett-Packard's technical systems sector, on loan to OSF. He is now the permanent vice-president of research and development at OSF. John Paul is the vice-president and director of software development for Nixdorf Computer, also on loan to the OSF. Alex Morrow is the director of strategic alliances, membership recruitment, and international communications. He is on loan to the OSF from IBM technical computing systems, where he is a manager of systems architecture and development. This meeting was the first time, and to date the only time, the OSF management team had met with technical Unix users and developers.

UNIX vs. UNIX

During the information meeting with the USENIX Association conference attendees, Goldstein, Paul, and Morrow provided general information on the goals of the OSF and how these goals would be achieved. A question-and-answer period followed the presentation. Answers to some of the more technical questions were not provided by the speakers, however.

Many conference attendees left the meeting with the belief that the OSF was not doing anything concrete and that no plans or schedules were available. This meeting was held, however, only six weeks after the formation of the Open Software Foundation.

Another concern of conference attendees was that all the OSF managers were on loan from sponsor companies. Attendees were unsure if the OSF had any technical employees. Now, several months later, the foundation is recruiting technical personnel, but many in the community are reluctant to join the OSF because potential employees are unsure of what they would be working on, what the goals would be, and what the future holds for their employer.

The OSF has made some statements about how it expects to be working. It expects to have several hundred employees, including technical staff. It plans to foster research by providing grants to universities. It is also requesting technologies from both industry and university sources, and these technologies are expected to make up any OSF product offering.

First RFT

The first request for technology (RFT) was issued by the OSF on July 18, 1988. The RFT was for a graphical user interface technology to be incorporated into the OSF User Environment Component of the Application Environment Specification. Several companies responded,

including AT&T and Sun, which submitted their *Open Look* interface.

The OSF is planning to establish a user interface standard to allow applications to be ported easily from different platforms and to encourage appropriate levels of user interface consistency between applications. The RFT requires that interfaces submitted meet the following criteria:

- Conform with the IEEE POSIX standard and the MIT X Window System, a public-domain windowing-system standard created by the Massachusetts Institute of Technology
- Are written in ANSI C and support applications written in ANSI C, without precluding other OSF programming languages (no languages are specified)
- Are portable across a wide range of computers
- Can be shipped commercially during the first half of 1989
- Can support a broad range of national languages, including European, Semitic, and Asian languages
- Include provisions for testing facilities to prove the above claims
- Contain reasonable and equitable licensing terms

Submissions to the RFT were due in mid-September. Both sponsors and members of the OSF will select the winning interface technology in meetings scheduled for November of this year.

Licensing Concerns

The OSF's RFT has provoked another concern among vendors and users in the Unix industry. The RFT is open to any company or university. Consequently, the OSF's final product could be constructed of pieces from several different sources, which brings up licensing concerns. Any Unix product from the OSF will have an AT&T license, as AT&T owns Unix. OSF has purchased an IBM AIX license and has stated that it will use AIX as the base of its operating system offering. Another license will be required by the university or company that contributes the graphical user interface. If other RFTs are issued, as expected, other licenses will be required. All these licenses could add up to a substantial sum of money for users. The hodgepodge of license terms could be a constraint on further innovation in the Unix operating system for companies licensing the OSF product.

Licensing problems were first apparent in the Unix community with the release of AT&T's System V, Release 3, several years ago. In the past, AT&T had put few licensing restrictions on its licensees, including the earlier System V, Release 2, system. The System V, Release 3, Unix license contained significant constraints, however. AT&T was trying to gain more control over Unix and limited how licensees could rewrite or enhance Unix for their product offerings. AT&T also specified that the Unix products must be able to pass an AT&T test suite. Bugs have been reported in the AT&T test suite, however, that limit the ability of Unix variations to pass the tests. Many companies are refusing to purchase System V, Release 3, and are still using Release 2.

The OSF came up against this problem when it purchased an AT&T Unix operating system license for System V, Release 3. It is aware, however, that many companies will not purchase a product with a Release 3 license. The OSF is currently unsure as to how it will handle this problem when it has a product ready for market.

POSIX will not solve all incompatibility problems, but it will help

AT&T has recently chosen to spin off its Unix operating system into a separate division of AT&T. The OSF was founded by companies concerned that AT&T and Sun would have complete control over Unix. AT&T, in creating this separate division, may deflect the fears that AT&T will have complete control over Unix and dictate product offerings to the companies and universities offering Unix as a product.

AT&T has also announced another major change. On October 18, 1988, AT&T announced for formation of a new group to guide AT&T System V Unix. Talks with OSF had broken down over OSF's requirement that their offering be based on IBM's AIX, a position unacceptable to AT&T. The new group of AT&T supporters, including several major companies, is expected to offer guidance to AT&T in software development and licensing terms. No plans for a Unix separation from AT&T have been announced as yet, however.

AT&T is now licensing the name Unix for corporate versions of the operating system, so companies can license the name for their versions of Unix. In the case of Unix for the PC market, AT&T and Microsoft have merged AT&T's System V, Release 3, and Microsoft's Xenix into a single PC Unix operating system. The new system is called Unix SystemV/386 and was released to continue AT&T's efforts to create a standard Unix. The Santa Cruz Operation (SCO) release of its newest version of Unix will be named SCO Unix System V/386 and is set for a mid-1989 release.

AT&T has stated that it plans to continue licensing the name Unix to other companies. This additional income would allow AT&T to consider relinquishing some of its control over Unix in order to promote a more unified Unix product.

A Single Standard

AT&T has stated that it wishes to see a single Unix standard. Discussions between AT&T and the OSF may produce an agreement by the end of this year. Sun has also expressed interest in creating a single, unified Unix operating system and is also considering working with the OSF to create a single OSF and AT&T/Sun standard Unix.

The OSF has made participation in the development of future Unix releases a condition of any AT&T/OSF agreement. If an agreement is reached and a single Unix operating system is created, with both the OSF and AT&T distributing it, both organizations would receive licensing fees from the distribution. The Open Software Foundation, whether it produces a product or not, may in the long run help the Unix market. It may eventually push AT&T into creating a single Unix operating system standard that does not inhibit the ability of companies to enhance their own versions of Unix. AT&T and Sun were originally planning to create a standard Unix. It may be that a single Unix is created but that AT&T, Sun, and the sponsors and members of the OSF may decide on control of development.

It remains to be seen, however, whether the OSF really does help the Unix market. If the OSF and AT&T/Sun cannot come to an agreement, the market may remain split with several major forms of Unix, including AT&T, OSF, and Berkeley.

Users and developers, to help themselves during this transition period, have the IEEE POSIX standard to guide them. Right now, though, the community is watching and waiting to see how the situation unfolds over the next two years.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Writing OS/2 Applications with I/O Privileges

Segregating hardware-dependent code into IOPL modules lets you bypass OS/2 to take control of peripheral devices

by Ray Duncan

The advent of OS/2, Microsoft's new protected-mode, multitasking, virtual-memory operating system for 80286-based microcomputers, has been accompanied by epic amounts of confusion, misinformation, and disinformation among programmers and computer journalists. A particularly popular misconception is that the existence of the Unix system makes OS/2 unnecessary, a notion that is vigorously promoted by the Unix partisans who sense that yet another opportunity to conquer the desktop is slipping from their grasp. In reality, Unix and OS/2 should not be viewed as competitors at all: They are systems with completely different goals, characteristics, and market niches.

The Unix system was designed to support multiuser applications, and its enforcement of interprocess and interuser protection is strict. Users' access to files, directories, and programs is restricted with an elaborate set of passwords, permissions, and administrative controls —all of which have a sizable memory and disk overhead. Direct manipulation of the hardware by application programs is strictly forbidden because this would wreak havoc in a multiuser environment; only device drivers may access peripheral devices, and a new device driver cannot be installed without relinking the operating system kernel. Even the programming environment is remarkably inflexible: writing Unix applications in anything but C or a language that was in turn written in C is nearly impossible because the C run-time library is inextricably interwoven with Unix kernel services. In most Unix systems, the actual interface to the kernel is usually documented only sketchily or not at all.

OS/2, on the other hand, was designed as a single-user operating system, so its philosophy of protection is vastly different. OS/2 is a completely "open" system; there are no passwords and permission schemes, the interface to the kernel services is clean and well documented, and custom device drivers are easily installed by copying them to the boot disk and adding a line to the system configuration file. Because users are expected to be responsible for both the programs they install into their system and (if necessary) for securing physical access to the system, OS/2 allows applications a degree of freedom that would be unthinkable in a multiuser system such as Unix. Programs can, for example, generate machine code dynamically in a data segment and then execute it, manipulate device adapters directly without resorting to a device driver, and filter the raw data stream of the keyboard, mouse, and printer drivers.

The ability to bypass OS/2 and take control of a peripheral device is particularly important for graphics applications that are not written to run in a Presentation Manager window or that rely on video display modes or capabilities that are not supported by OS/2's built-in video drivers. OS/2 places only three constraints on such programs: They may not service interrupts (control of the interrupt system is reserved to the kernel and true device drivers), they must cooperate with OS/2 to save and restore the screen and adapter state across session switches, and the hardware-dependent code must be segregated into special modules called I/O privilege level (IOPL) segments —which I will explain in this article.

Ray Duncan is a software developer in Los Angeles, Calif. You can reach him at P.O. Box 10420, Marina Del Rey, CA 90295.

80286 Protection Mechanisms

In order to understand IOPL segments, we first need to make a small digression into the architecture of the Intel 80x86 family of processors. The 80286's protection facilities are based on two key concepts: segment selectors and privilege levels.

All the CPUs in the Intel 80x86 family generate memory addresses by combining the contents of segment registers (which may be thought of as base pointers) with an offset or displacement from the machine instruction and/or one or more index registers. In the case of the 80286, the hardware's interpretation of a value in a segment register depends on whether the processor is running in real mode or protected mode.[1]

In real mode, which is essentially an 8086/88 emulation mode, the hardware provides no protection mechanisms; any program can read or write any memory address or access any I/O port. The values in segment registers are paragraph addresses (20-bit physical addresses divided by 16); to form a complete memory address, the CPU shifts the contents of a segment register left 4 bits and adds it to a 16-bit offset (see Figure 1, page 37). MS-DOS and its applications execute in real mode, regardless of the type of processor.

In protected mode, a level of addressing indirection is added. Segment registers do not point directly to the base of an area of memory; instead, they contain values called selectors. A selector is an index to an entry in a descriptor table; the entry contains a memory segment's base address, length,

and other characteristics. Each time a program references memory, the hardware uses information from the descriptor table to generate the physical address and simultaneously validates the memory access (see Figure 2, this page).

Because only the operating system can manipulate descriptor tables, and these tables govern the addressable memory space of all programs, a protected-mode operating system can completely isolate programs from one another. If a program attempts to read or write memory that does not belong to it, or call a routine to which it has not been given access, a CPU fault (similar to a hardware interrupt) occurs and the operating system regains control—this is the meaning of the term *memory protection.*

The behavior of programs in protected mode is also constrained by the 80286's support for four privilege levels, called ring 0, ring 1, ring 2, and ring 3. Programs running at ring 0 have unrestricted access to the hardware, including the ability to execute any instruction, read or write any I/O port, and manipulate the special registers and tables that control memory protection and virtual memory.

Rings 1 to 3 are intended for the execution of progressively "less trusted" programs. Transitions from one ring to another are strictly controlled by means of "call gates," which can only be set up by a program running at ring 0. Programs in rings 1 and above cannot execute certain instructions that would compromise memory protection or touch memory segments for which they are not qualified (one of the characteristics in a memory segment's descriptor is the privilege level required for access to the segment).

Programs in rings 1, 2, or 3, however, may gain restricted access to the hardware depending on the value of the IOPL field in the CPU flags register. Whenever a program is running in a ring whose number is less than or equal to the contents of the IOPL field, it can use the six instructions *IN, OUT, INS, OUTS, STI,* and *CLI* without causing a protection fault. Needless to say, the IOPL field itself can only be modified by a program running at ring 0.

Under OS/2, ring 0 is reserved for the operating system kernel and its device drivers. Ring 1 is not used at all, and normal application code runs at ring 3. The operating system sets the IOPL level to 2 and uses ring 2 only for the execution of code within application IOPL segments (see Figure 3, this page). The names of the IOPL segments, along with the names of the routines within them that will be accessible to the rest of the application, must be declared at link time.

When an application containing an IOPL segment is loaded, OS/2 sets up call gates for each "exported" routine in that segment and resolves the references to those routines throughout the program to point to the call gate. When the program's mainline code, which is running at ring 3, calls a routine in an IOPL segment, the hardware traps the reference to the call gate. The CPU then changes the privilege level to ring 2, switches to a new stack, copies pa-

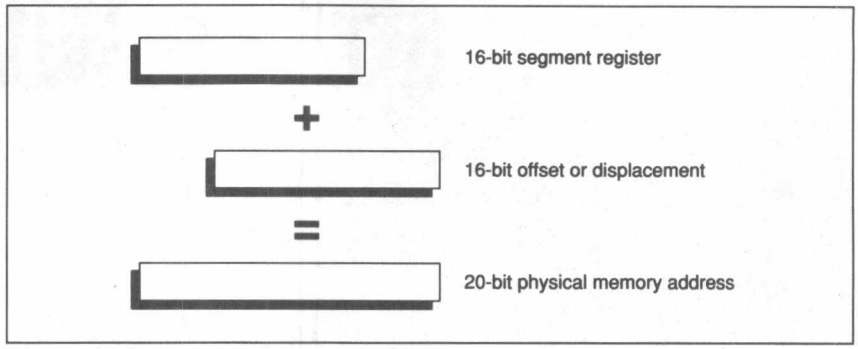


Figure 1: *Generation of memory addresses in real mode*

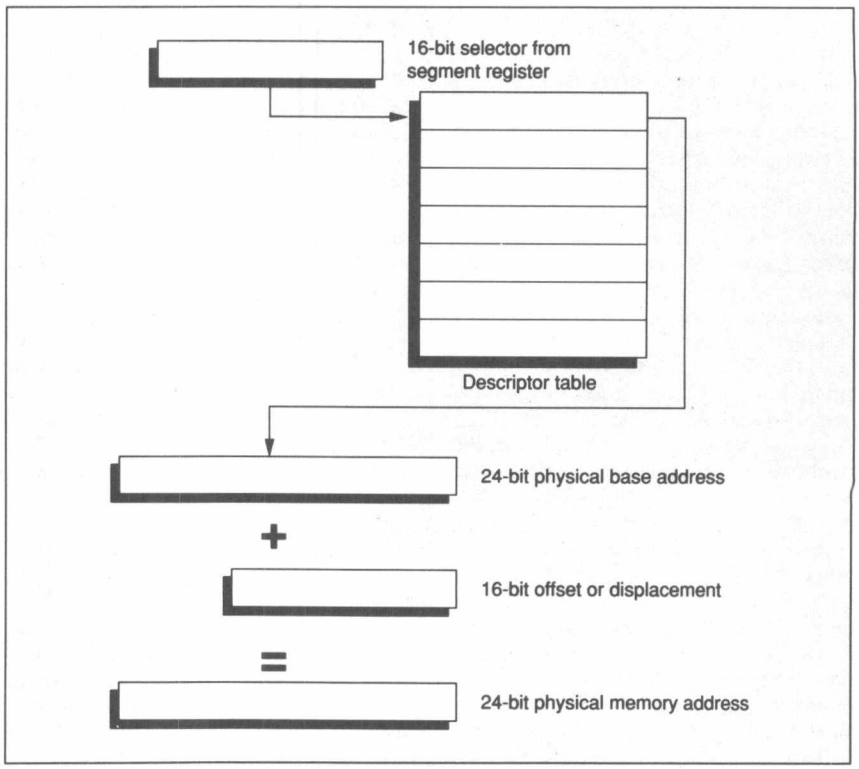


Figure 2: *Generation of memory addresses in protected mode*

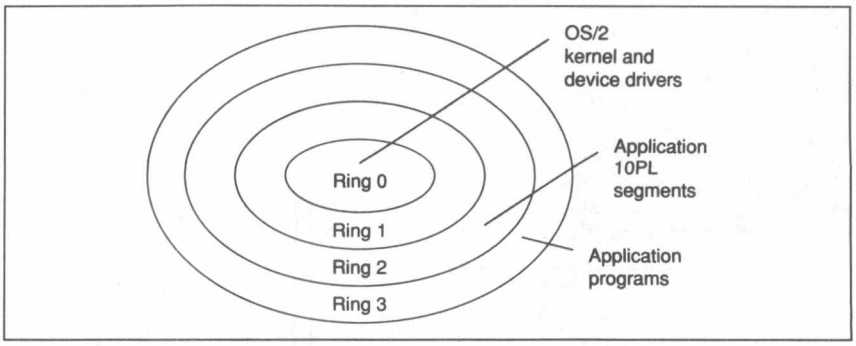


Figure 3: *Use of protected-mode privilege levels in OS/2*

rameters from the old stack to the new one, and enters the IOPL routine. When the IOPL routine exits with a *far* return, the original privilege level and stack are restored, the parameters are cleared from the caller's stack, and execution continues at ring 3.

Application programs with IOPL, like real-mode MS-DOS programs running in OS/2's DOS compatibility environment, are a weak point in system security. It is easy to imagine, for example, an OS/2 protected-mode "virus" program that would masquerade as a public-domain utility but would use an IOPL segment to take over the disk controller and scribble all over the disk directories and file allocation table. To prevent such disasters, OS/2 can be configured so that it will not allow execution of MS-DOS programs and/or protected-mode programs requiring IOPL. If the CONFIG.SYS file contains the directive *PROTECTONLY=YES*, MS-DOS applications are not supported; if the directive *IOPL=NO* is present, applications with IOPL segments will not be loaded.

The ability to bypass OS/2 and take control of a peripheral device is particularly important for graphics applications that do not use the Presentation Manager

Using IOPL in OS/2 Applications

Designing and coding an OS/2 application that uses IOPL segments is straightforward, if you follow just a few simple rules.

All code that needs to enable or disable interrupts or access I/O ports must be segregated from the rest of the application code into a distinct segment that will become the IOPL segment. Calls to OS/2 application program interface (API) functions from within the IOPL segment should be avoided. For compatibility with Microsoft C, the segment should have the *WORD* and *PUBLIC* attributes and be assigned to class *CODE*. Be sure to pick a segment name that won't conflict with the standard Microsoft segment and group names (*_TEXT*, *_DATA*, *DGROUP*, and so forth).

IOPL SEGMENTS

To make the procedures in the IOPL segment usable from a high-level language, they should follow the OS/2 API conventions of accepting their parameters on the stack and return their results in register *ax* or in variables whose addresses were passed in the original call. The procedures that contain the hardware-dependent code must be declared *PUBLIC* and given the *far* attribute because they will be entered through a call gate and must exit with a *far* return. The parameter to the *RET* instruction must be the number of bytes (not words) to be cleared from the caller's stack.

If you are writing the body of your application in C, you should supply *extern far pascal* prototypes for the external IOPL routines. This tells the C compiler that parameters are pushed left to right, the called routine clears the stack, a leading underscore should not be added to the external name, and case in the external name can be ignored. If you are writing the application with MASM, you simply declare the IOPL routines as *extrn far* in the normal manner.

Second, the initialization portion of your application should include a call to one of the kernel API functions *DosPortAccess* or *DosCLIAccess*. *DosCLIAccess* informs the operating system that the application will be using the *CLI* and *STI* instructions to disable and enable interrupts. *DosPortAccess* notifies the operating system of a range of I/O ports to be used by the application; *DosPortAccess* also implicitly grants *CLI* and *STI* access, and an additional call to *DosCLIAccess* is not required.

In the current versions of OS/2, *DosPortAccess* and *DosCLIAccess* do nothing; an application with IOPL can read or write I/O ports and execute *CLI* or *STI* whether it calls these functions or not. *DosPortAccess* and *DosCLIAccess* are present for upward compatibility with future, 80386-specific versions of the operating system. The 80386 allows access to individual I/O ports to be controlled on a per-process basis with an I/O permissions bitmap associated with each task state segment (TSS).

Third, when you build the executable application, you must provide the linker with a module definition (.DEF) file. Module definition files describe application type, segment behavior, and imported or exported routines, among other things. If you are writing your program in C, you will need to compile and link as separate operations because the C compiler does not know how to pass the name of a module definition file through to the linker.

In the case of an IOPL application, the .DEF file must contain a *SEGMENTS* directive that applies the IOPL attribute to the appropriate segment name. It must also contain an *EXPORTS* statement that declares the name and number of stack parameters for each routine in the IOPL segment that will be called from outside the segment. This information is built into the .EXE file and is later used by the OS/2 loader to set up the necessary call gates.

If you fail to provide the *EXPORTS* declaration, the IOPL routines are entered directly instead of through a call gate that changes the privilege level, and the program is terminated with a protection fault when it first executes an *IN*, *INS*, *OUT*, *OUTS*, *CLI*, or *STI* instruction. If the *EXPORTS* statement is present but too few parameters are specified for an IOPL routine, the program will likely run into a protect fault when it tries to access a stack parameter that isn't there. If too many parameters are specified, no harm is done unless the parameter to the *RET* instruction in the IOPL routine is also wrong; in that case, too many words will be cleared from the caller's stack when the IOPL routine exits, usually resulting in a protection fault at some later point in the program's execution.

An Example IOPL Application

To provide a practical demonstration of an OS/2 application that uses direct hardware access, I have written a simple program called PORTS.EXE that reads and displays the first 256 I/O

```
        title   PORTIO.ASM  read/write I/O ports
        page    55,132
        .286
; PORTIO.ASM -- general purpose port read/write
;             routines for C or MASM programs
; Copyright (C) 1988 Ray Duncan
; When this module is linked into a program, the
; following lines must be present in the program's
; module definition (.DEF) file:
;
; SEGMENTS
;    IO_TEXT IOPL
;
; EXPORTS
;    rport 1
;    wport 2
;
; The SEGMENT and EXPORT directives are recognized by
; the Linker and cause information to be built into
; the .EXE file header for the OS/2 program loader.
; The loader is signalled to give I/O privilege to
; code executing in the segment IO_TEXT, and to build
; call gates for the routines 'rport' and 'wport'.

IO_TEXT segment word public 'CODE'

        assume  cs:IO_TEXT


; RPORT: read 8-bit data from I/O port.  Port address
; is passed on stack, data is returned in register AX
; with AH zeroed.  Other registers are unchanged.
;
; C syntax:     unsigned port, data;
;               data = rport(port);

        public  rport
rport   proc    far

        push    bp              ; save registers and
        mov     bp,sp           ; set up stack frame
        push    dx

        mov     dx,[bp+6]       ; get port number
        in      al,dx           ; read the port
        xor     ah,ah           ; clear upper 8 bits

        pop     dx              ; restore registers
        pop     bp

        ret     2               ; discard parameters,
                                ; return port data in AX
rport   endp


; WPORT: write 8-bit data to I/O port.  Port address and
; data are passed on stack.  All registers are unchanged.
;
; C syntax:     unsigned port, data;
;               wport(port, data);

        public  wport
wport   proc    far

        push    bp              ; save registers and
        mov     bp,sp           ; set up stack frame
        push    ax
        push    dx

        mov     ax,[bp+6]       ; get data to write
        mov     dx,[bp+8]       ; get port number
        out     dx,al           ; write the port

        pop     dx              ; restore registers
        pop     ax
        pop     bp

        ret     4               ; discard parameters,
                                ; return nothing
wport   endp

IO_TEXT ends

        end
```

Example 1: *PORTIO.ASM, the source code for the IOPL segment that contains the routines to read and write I/O ports*

```
NAME PORTS WINDOWCOMPAT

PROTMODE

SEGMENTS
  IO_TEXT IOPL
EXPORTS
  rport 1
  wport 2
```

Example 3: *PORTS.DEF, the module definition file for the PORTS.EXE demonstration program*

```
ports.obj : ports.c
  cl /c ports.c

portio.obj : portio.asm
  masm /Mx portio;

ports.exe : ports.obj portio.obj ports.def
  link ports+portio,ports,,,ports.def
```

Example 4: *The MAKE file for PORTS.EXE*

ports. PORTS.EXE is built from three modules: PORTIO.ASM, PORTS.C, and PORTS.DEF.

PORTIO.ASM (Example 1, page 42) is the program's IOPL segment. It contains only two routines: *rport* and *wport*. *rport* accepts a port address, reads the port, and returns the port data register ax with the upper 8 bits zeroed. *wport* accepts a port address and a word of

```
[C:] PORTS  <Enter>

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  00 00 00 00 AC FF 00 00 00 FF FF FF FF 00 FF FF
0010  00 00 00 00 AC FF 00 00 00 FF FF FF FF 00 FF FF
0020  00 A9 00 A9 00 A9 00 A9 00 A9 00 A9 00 A9 00 A9
0030  00 A9 00 A9 00 A9 00 A9 00 A9 00 A9 00 A9 00 A9
0040  50 06 7C FF 96 07 00 FF 08 FF BA FF 25 FF 76 FF
0050  C0 0C 7C FF 41 0C 00 FF 98 FF C4 FF 9B FF 6D FF
0060  9C 20 9C 30 1C 20 1C 30 FF FF FF FF FF FF FF FF
0070  FF 00 FF 00 FF 00 FF 00 FF FF FF FF FF FF FF FF
0080  00 13 0F 0F BC 00 62 0F FF 0F 0F 0F 00 FF 00 00
0090  00 13 0F 0F BC 00 62 0F FF 0F 0F 0F 00 FF 00 00
00A0  00 DC 00 DC 00 DC 00 DC 00 DC 00 DC 00 DC 00 DC
00B0  00 DC 00 DC 00 DC 00 DC 00 DC 00 DC 00 DC 00 DC
00C0  1C CB E4 34 00 00 00 00 00 00 00 00 00 00 00 00
00D0  00 00 FF FF FF FF FF FF FF FF 00 00 FF FF FF FF
00E0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00F0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

[C:]
```

Example 5: *Example of the output of PORTS.EXE*

```
/*
    PORTS.C:    Demonstration program with IOPL.
                Reads and displays the first 256 I/O ports.
                Requires separate module PORTIO.ASM.

    (C) 1988 Ray Duncan

    To build:   MASM /Mx PORTIO;
                CL /c PORTS.C
                LINK PORTS+PORTIO,PORTS,,,PORTS.DEF

    Usage:      PORTS
*/

#include <stdio.h>

#define API extern far pascal

unsigned API rport(unsigned);           /* function prototypes */
void     API wport(unsigned, unsigned);
void     API DosSleep(unsigned long);
unsigned API DosPortAccess(unsigned, unsigned, unsigned, unsigned);

                                    /* parameters for DosPortAccess */
#define REQUEST 0                       /* request port */
#define RELEASE 1                       /* release port */
#define BPORT   0                       /* beginning port */
#define EPORT   255                     /* ending port */

main(int argc, char *argv[])
{
    int i;                              /* scratch variable */

                                        /* request port access */
    if(DosPortAccess(0, REQUEST, BPORT, EPORT))
    {
        printf("\nDosPortAccess failed.\n");
        exit(1);
    }

    printf("\n      ");                  /* print title line */
    for(i=0; i<16; i++) printf(" %2X", i);

    for(i=BPORT; i<=EPORT; i++)         /* loop through all ports */
    {
        if((i & 0x0f)==0)
        {
            printf("\n%04X  ", i);      /* new line needed */
        }

        printf(" %02X", rport(i));      /* read & display port */
    }
                                        /* release port access */
    DosPortAccess(0, RELEASE, BPORT, EPORT);
}
```

Example 2: *PORTS.C, the source code for the body of the PORTS.EXE example program*

data, writes the low 8 bits of the data to the port, and returns nothing. These routines both use the OS/2 API conventions and can be called from either MASM or C. Notice that the code segment in this file is called *IO_TEXT* to differentiate it from the code segment produced by the C compiler (which has the default name *_TEXT*).

PORTS.C (Example 2, page 43) is the body of the application. It invokes *DosPortAccess* to request access to a range of I/O ports and then calls *rport* to read each port, formatting the data for display with *printf*. Before terminating, the program again calls *DosPortAccess* to release the I/O ports requested earlier.

PORTS.DEF (Example 3, page 43) is the module definition file. The *NAME* statement causes the linker to build an executable program rather than an OS/2 dynamic link library or device driver and also states (with the *WINDOWCOMPAT* option) that the program can run in a Presentation Manager window. The *SEGMENTS* directive marks *IO_TEXT* as an IOPL segment. The *EXPORTS* directive defines the number of stack parameters for *RPORT* and *WPORT* and that they are callable from outside the IOPL segment.

To build the executable program PORTS.EXE from the source files PORTS.C, PORTIO.ASM, and PORTS .DEF, enter the following sequence of commands:

```
[C:\] CL /c PORTS.C
[C:\] MASM /Mx PORTIO;
[C:\] LINK PORTS+PORTIO,
                PORTS , , , PORTS.DEF;
```

You need not specify any libraries in the *LINK* command because they are taken care of by "default library" records embedded in the PORTS.OBJ file, but make sure that the reference library OS2.LIB is available in one of the directories specified in the environment *LIB=* string. You can also automate the construction of PORTS.EXE with the MAKE utility; the MAKE file is shown in Example 4, page 43.

When you run PORTS.EXE, you should see output in the form shown in Example 5, this page. If you get the error message "OS/2 is not configured to run this application," add the statement *IOPL=YES* to your CONFIG.SYS file and reboot the system.

Note

1. The 80386 has three different protected modes, but the current versions of OS/2 run on the 80386 as though it were a 80286.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Writing Programs for *MultiFinder*

If you're writing a new Mac application or revising an old one, it pays to make it MultiFinder aware

by Chris Derossi

MultiFinder, Apple's multitasking extensions to the Mac OS, was first introduced with System 4.2 and was later enhanced with System 6.0. Although the use of MultiFinder is now familiar to most, programmers writing for the MultiFinder environment still find themselves faced with some confusion. In this article, I'll address some of the issues relating to programming for MultiFinder and hopefully clear up much of the confusion.

Sometimes programming for the Macintosh is like trying to shoot at a moving target. As the Mac system software changes, so do the rules for writing "well-behaved" programs. This is part of the reason why programming the Mac today seems to be extra-complicated.

Prior to System 4.2 (pre-MultiFinder), each application for the Mac effectively owned the entire machine during its execution. There were guidelines then for writing programs that were considered well-behaved and compatible, but they were just that —guidelines. In the interest of getting just a little bit more functionality or a little bit more performance, these rules were sometimes bent or simply ignored.

Enter MultiFinder. Not only did MultiFinder tend to break programs that broke the rules, but also the rules themselves changed. Programs had to learn to run in a world in which everything —such as the file system, memory, events, and the screen —was shared.

Chris Derossi is a member of the Apple Developer's Technical Support Team and can be reached at 20525 Mariani Ave., MS-51T, Cupertino, CA 95014.

At the same time as programs were being revised to become MultiFinder compatible, a new class of programs was born. These MultiFinder-aware applications were designed to take advantage of a multitasking environment. Such programs could do background processing and deal intelligently with being suspended and resumed.

With System 6.0 some new features were added to MultiFinder. Applications being written or revised today have a different set of guidelines to follow than did programs written for MultiFinder and System 4.2. Such is the price we pay to have software that evolves with the times. Of course, the changes between System 4.2 and System 6.0 are less dramatic than the changes that went along with MultiFinder's introduction.

One of the things that has changed is the *SIZE* resource. It was originally defined so that Switcher could know something about applications, such as the amount of memory they need. MultiFinder adopted the *SIZE* resource, ignoring the Switcher flags and defining new flags. Each flag in the *SIZE* resource tells MultiFinder something about how the application wants to be executed or how much the application understands.

The most recent *SIZE* resource definition is shown in Figure 1, page 47; the corresponding *Rez* type description is in Example 1, page 47.

What the Flags Mean

The bits in the flags field have been cause for some confusion for several reasons. First, it isn't always clear exactly what each flag means or what the results are of setting each one. Also, even though each bit can be set independently of the others, there is a relationship between some of the flags that may not be obvious. Finally, some of these flags have had different names at different times.

Here is a description of each flag along with the effects of setting or not setting that flag. These descriptions are in the order that you would most likely want to consider them for your application.

acceptSuspendResumeEvents — This flag tells MultiFinder whether or not your application understands and will handle suspend and resume events. If this flag is set, MultiFinder will issue your application a suspend event before you are switched into the background and a resume event once you have been switched to the foreground.

On a suspend event, MultiFinder will expect you to convert your private scrap to the Clipboard. You don't have to do anything if your private scrap hasn't changed since the last resume (or since your application was started). On a resume event, your application has to convert the Clipboard to your private scrap. You only have to do this if the Clipboard has changed while you were suspended. If the Clipboard has changed, bit 1 will be set in the resume event's message field.

If this flag is not set, then MultiFinder acts somewhat differently. Your application won't get suspend and resume events, so it won't know that it has been switched out. It also won't know when it should convert its private scrap to the Clipboard. Because of this, Multi-

Finder has to fool your application into converting its scrap.

Because your application would normally convert its scrap only when a desk accessory was opened, MultiFinder pretends to open a desk accessory. This is done by feeding a mouse-down event in the menu bar to the application. The application then calls *MenuSelect* and is passed back the item number of the *About MultiFinder . . .* item in the Apple menu. Thinking it's a DA, your application calls *OpenDeskAcc* on *About MultiFinder . . .* and converts its scrap.

multiFinderAware —This flag is used only if the *acceptSuspendResumeEvents* flag is also set. Unlike Switcher, MultiFinder keeps all application windows on the screen. This means that the front window of suspended applications needs to be deactivated to present the user with a consistent interface. If this flag is set, MultiFinder will expect the application to deactivate and activate its front window when it receives suspend and resume events. This is why this flag used to be called *doOwnActivate*.

Activating and deactivating windows entails highlighting/unhighlighting any selected text, showing/hiding scroll bars, calling *TEActivate/TEDeactivate*, and so on. These are the same actions as would be done in response to normal activate and deactivate events.

If this flag is not set, then MultiFinder must fool your application into activating and deactivating its front window. MultiFinder does this by feeding your application the appropriate activate or deactivate event.

Most straightforward applications written today should have both the *acceptSuspendResumeEvents* and *multiFinderAware* flags set. These two flags basically determine whether the responsibility for handling suspension and resumption rests with the application or with MultiFinder. Remember that if these flags are not set, MultiFinder has to fool your application into converting the scrap and activating/deactivating its front window.

Your application is doing the real work anyway. The difference is that if MultiFinder has to trick your application, several extra steps are involved. This takes time. If your application handles everything with a single suspend or resume event, switching between applications is much faster.

canBackground —If this flag is set, MultiFinder will give null events to your application while it is in the background. The frequency and number of null events you receive is dependent on the sleep value that you pass to *WaitNextEvent* and the amount of time given up by the foreground application.

MultiFinder's first priority is the foreground application's responsiveness. Because some older applications relied on getting frequent null events, if the application in the foreground is calling *GetNextEvent* instead of *WaitNextEvent*, it will get many of the null events. On the other hand, if the foreground application is polite and calls *WaitNextEvent* with a high sleep value, background applications will get plenty of time.

You should set this bit only if you really have something to do in the background. If your application doesn't do anything when it gets a null event (or if it just changes the cursor, for example), then don't set this bit. Asking for null events when your application doesn't use them steals time needlessly from other applications.

getFrontClicks —If the user brings your application to the front by clicking on one of its windows, the mouse-down event is normally used to resume your application only. If this bit is set, though, your application will get

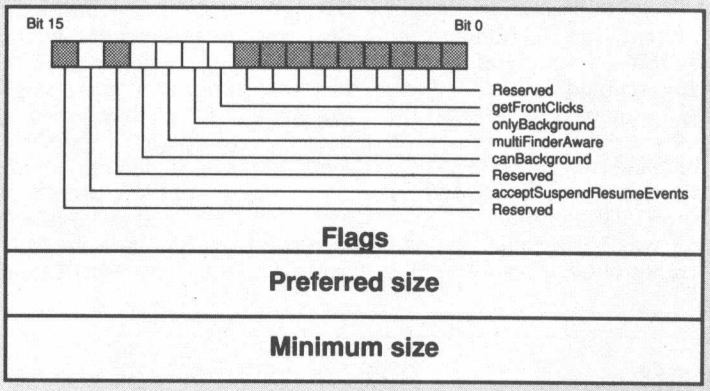


Figure 1: *The SIZE resource definition*

```
type 'SIZE' {
  boolean     dontSaveScreen,       /* No longer used */
              saveScreen;

  boolean     ignoreSuspendResumeEvents,
              acceptSuspendResumeEvents;

  boolean     enableOptionSwitch,   /* No longer used */
              disableOptionSwitch;

  boolean     cannotBackground,
              canBackground;

  boolean     notMultiFinderAware,
              multiFinderAware;

  boolean     backgroundAndForeground,
              onlyBackground;

  boolean     dontGetFrontClicks,
              getFrontClicks;

  unsigned bitstring[9] = 0;        /* reserved     */

  /* Memory sizes are in bytes */
  unsigned longint;           /* preferred mem size */
  unsigned longint;           /* minimum mem size  */
};
```

Example 1: *The corresponding Rez type description*

(continued from page 47)

the mouse-down (and corresponding mouse-up) event that brought you to the foreground. This can be used to provide better responsiveness to the user.

For example, the Finder accepts these mouse-down events so you can click on an icon when the Finder is in the background. The Finder will come to the foreground and select the icon immediately, without your having to click on the icon a second time.

This type of action is not appropriate for all kinds of applications, however. If a paint program had this bit set, for example, just the act of bringing the application to the front might result in a dot being painted onto the document. The key here is to do what the user expects so that the user never realizes there are two alternatives.

onlyBackground —If this bit and the *canBackground* bit are both set, your application will be launched directly into the background. It cannot come to the foreground because it can't have any windows, its name will not show up in the Apple menu, and its icon will not be available in the menu bar.

Most applications will not have this bit set. Only if you are writing an unusual program should you have to deal with this flag. An example of the kind of application that uses this feature is Backgrounder. Backgrounder is launched by MultiFinder and stays in the background. Its sole job is to look for the existence of a spool file and launch PrintMonitor.

MultiFinder Tricks

In addition to tricking your application into converting its scrap and activating/deactivating its front window, there are two other times when MultiFinder will fool your application into doing something. One of these happens when the user chooses *Restart* or *Shutdown* from the Finder's Special menu. MultiFinder goes through all the currently executing applications and makes them think that the user has chosen *Quit*. This causes each application to terminate, asking the user to save documents as appropriate for each application.

The other instance of MultiFinder tricking the application happens when an application is already running and the user launches a document for that application from the Finder. MultiFinder switches to that application and posts a mouse-down in the menu bar, selecting the *Open . . .* item from the File menu. Then, when the application calls *SFGetFile* in response to the *Open . . .*, MultiFinder simply returns a reply for the launched document.

This could fail for a couple of reasons. If your application doesn't call *SFGetFile* or *SFPGetFile* in response to the *Open . . .*, the user will get your application but the document won't have opened. If for some reason a document cannot be opened, your application should gray the *Open . . .* item (for example, you support only one document at a time and one is already open).

A far more common reason why MultiFinder's quit or open tricks would fail is nonstandard *Quit* or *Open . . .* menu items. MultiFinder looks for a

menu with the title "File" and items named *Quit* for quitting and *Open* (with ellipses), *Open . . .* (with three periods), *Open . . .*, or *Open Stack* for opening documents. If these are not present, the user will have to quit or open documents manually from the application.

If you have nonstandard menus in your application that perform the same action as the standard ones, there is a way to tell MultiFinder what your menu items are called. MultiFinder looks for certain resources in your program that contain the names of your *Quit* and *Open* menu items. These resources are of type *'mstr'* and are equivalent to the standard *'STR'* resource type. Because your *Quit* and *Open* items may be in different menus, each of them have a *'mstr'* resource for their menu title and a *'mstr'* resource for their item name. MultiFinder expects the menu title for the *Quit* item to be in *'mstr'* #100 and the menu item to be in *'mstr'* #101. Similarly, the *'mstr'* resources for the *Open* item are #102 and #103.

Figure 2, this page, shows a nonstandard File menu, Example 2, this page, shows the *Rez* source for the corresponding *'mstr'* resources.

If, for some reason, your application uses multiple strings for the *Quit* or *Open* menu items, then you can list all the possibilities using *'mst#'* resources instead of *'mstr'* resources. The *'mst#'* resource type is the same as the *'STR#'* resource type, which contains a list of strings. Example 3, this page, shows the *Rez* source using *'mst#' resources.*

Background Only Applications

Historically, programs that needed to do some work periodically but that didn't have any user interaction would have to be written as drivers that had

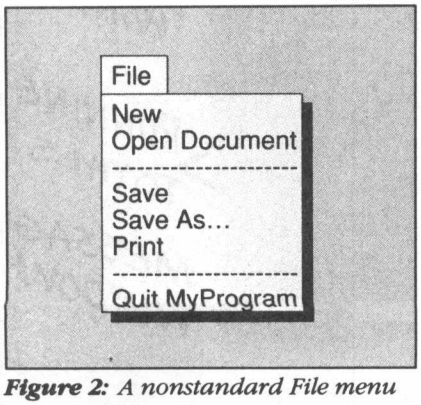


Figure 2: *A nonstandard File menu*

```
resource 'mstr' (100) {"File"};

resource 'mstr' (101) {"Quit
MyProgram"};


resource 'mstr' (102) {"File"};

resource 'mstr' (103) {"Open
Document"};
```

Example 2: *The Rez source for the corresponding 'mstr' resources*

```
resource 'mst#' (101) {
 {
   "File";
   "Files"
 }
};

resource 'mst#' (102) {
 {
   "Quit MyProgram";
   "Quit To Finder"
 }
};
```

Example 3: *The Rez source using 'mst#' resources*

the *dNeedsTime* flag set. Examples of this kind of program are daemons, servers, and spoolers. But drivers have to be installed and can't have globals or multiple segments, so writing them could be cumbersome and difficult. MultiFinder now provides a better alternative.

By setting both the *canBackground* and *onlyBackground* bits in the *SIZE* resource, you can write an application that launches directly into the background and never comes to the front. This type of program is ideal for simple, periodic, background tasks. And background-only applications are easier to write than normal applications because they can't have any real user interface. This means that there are no windows, menus, controls, or dialogs to worry about. It also means that the only event that has to be dealt with is the null event; no user events are ever posted to this kind of application.

Plus these programs get all the advantages of being complete applications. Because they run normally and not at interrupt time, they can allocate memory and use handles. They have their own application partition, so they can have globals and multiple segments. And because they're not drivers, they don't have to be installed in the unit table; they can have icons and be launched just like any other application.

A background-only application doesn't have a layer of its own because it doesn't have any windows. Even if it tried to put up a window, dialog, or alert, it wouldn't succeed. But there is a way that such a program can communicate with the user if it needs to. This is one of the responsibilities of the Notification Manager.

Notification Manager

The Notification Manager allows a background application to display an alert, make a sound, or put an icon in the menu bar. In this rudimentary way, a background-only application can alert the user to error conditions, changes in the environment, or tasks completed.

Listing One, page 96, shows complete MPW Pascal source code for a background-only application. Listing Two, page 98, is the MPW C version, and Listing Three, page 100, is the *Rez* source for both versions. The main job that this application performs is comparing the current time to a specific time stored in the resource fork of the application. When the set time has been reached, an alert is displayed for the user. This program could be the beginning of a more general alarm-clock or appointment-reminder utility.

The flow of this sample program is simple. The first thing it does is initialize its world. Notice that the familiar Macintosh initialization calls (*InitGraf*, *InitFonts*, *InitWindows*, and so on) are missing. Because this program is not going to use any of the user interface managers, it doesn't have to initialize them. The init code sets two global flags to false and loads the alarm time and the text for the alert from the resource fork.

The main event loop is almost nonexistent because there are no events to handle. Instead, *WaitNextEvent* is used just to regulate background time. A large sleep value is passed to *WaitNextEvent* because this program doesn't need to get time very often and can afford to be polite. As soon as the current time (in minutes) has reached or passed the alarm time, the user needs to be notified.

While the alert is being displayed, the application will continue to get time in the background. For this reason, it has to be careful not to notify the user more than once. The first thing that the *TellUser* procedure does is check a global Boolean variable to see if the alert has already been requested. If the global *toldUser* is false, the Notification Manager is used to display an alert.

The Notification Manager queue element variable, *myNMBlock*, has to continue to exist after the *TellUser* procedure terminates, so it's a global variable. This is a background-only program that doesn't have an icon in the Apple menu, so the *nmMark* field is meaningless. For simplicity, only an alert is requested, and neither a sound nor an icon in the menu bar is used. Because the program needs to know when the alert has been dismissed, the address of a completion routine is passed in the *nmResp* field. After the fields have been filled in, *NMInstall* is called to initiate the notification.

At this point, the Notification Manager puts up an alert. This alert will come up on top of just about anything, including modal dialogs or other alerts that may currently be up.

As soon as the user dismisses the dialog, the completion routine, *AlertDone*, is called. Because this can happen at any time, the A5 world may belong to any running application when the completion routine is called. For this reason, the completion routine can't directly access any global variables or make intersegment calls.

The completion routine needs to do two things: It has to remove the notification element from the queue with *NMRemove*, and it has to set the global variable *doneFlag* to true so the application will terminate. Because of the restrictions on accessing global variables, a pointer directly to the *doneFlag* variable was saved in the *nmRefCon* field of the notification queue element. By using this pointer, the completion routine sets *doneFlag* to true.

The next time the program gets background time, it quietly terminates. A more comprehensive program might start looking for the next alarm time, or it might reset some flags and start waiting for the same alarm time the next day.

In Summary

If you are writing an application today, or revising an old application, it pays to make it MultiFinder aware. Very little extra code needs to be added, and the benefit of better speed during context switching is well worth it. Similarly, the inclusion of a *SIZE* resource, and '*mstr*' resources if appropriate, will help ensure that MultiFinder works well with your program.

Additionally, many programs will be able to take advantage of processing in the background. Background processing may be the edge your program needs to let your users get the most out of their Macs. Finally, the ability to have background-only applications opens the door for a whole new set of powerful spoolers, servers, and other background tasks.

Bibliography

Programmer's Guide to MultiFinder. APDA product #KMB017. Renton, Wash.: APDA.

MultiFinder Miscellanea. Macintosh Technical Note #180. Renton, Wash.: APDA.

Notification Manager. Macintosh Technical Note #184. Renton, Wash.: APDA.

MultiFinder Revisited. Macintosh Technical Note #205. Renton, Wash.: APDA.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 96.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Spelunking MS-DOS:
Documenting the Undocumented

Knowing what Microsoft doesn't tell you can help you write better TSRs

by Scott Robert Ladd

Spelunking —the art of exploring caves —requires a sense of adventure, a willingness to follow hunches, and a knowledge of how caves are constructed. Often the spelunker is entering uncharted territory, and must explore dead ends and dangerous areas. Finding the cavern filled with glistening crystals makes all the effort worthwhile.

Exploring the undocumented internals of MS-DOS is a process similar to spelunking. Debuggers and disassemblers are the tools, and working DOS utilities are the mountains containing the caves to be explored. You begin with a guess as to where you might find something interesting, and then follow a chain of logic from there. Sometimes you hit a dead end and have to back up and try a different path. Often, though, you find a cave filled with wonders —a function or feature of value.

This article details several useful, and sometimes essential, undocumented DOS functions and features I have uncovered while spelunking. Although I have confidence in my information, and have used these functions in my own programs, I recommend caution in using them. There are reasons why features are undocumented. In any version of MS-DOS some functions may disappear, or their behavior may change. Some OEM versions of MS-DOS may work differently than others. Using undocumented information can at times be dangerous, allowing you to manipulate or change certain aspects of the MS-DOS environment, which could lead to data loss or other problems.

So much for the obligatory warnings. In the text, I use the term *function* to refer to an MS-DOS service invoked through the INT 0x21 interrupt, and an *interrupt* is a specific MS-DOS software interrupt. I assume that you're familiar with MS-DOS, the Intel 80x86 family of microprocessors, and an assembler or a high-level language, which allows the programming of software interrupts (such as Turbo Pascal or C).

All of this information works in DOS, Versions 2.11, 3.10, 3.21, and 3.30. Because some DOS utilities (such as PRINT) use these capabilities, you can hope that things won't change with later versions of MS-DOS.

Scott Ladd is a freelance computer journalist and the C language columnist for Micro Cornucopia. *He can be reached at P.O. Box 61425, Denver, CO 80226.*

The Program Segment Prefix

Every time DOS loads a program, it creates a data structure called the program segment prefix (PSP). This is a 256-byte area located in memory just below the actual program. The PSP contains the program's command-line arguments, file tables, and other associated information. Although the PSP itself is documented, many of its fields are not. Table 1, page 57, shows the format of the PSP, including those undocumented fields I currently know of.

Let's look at each of the known fields individually:

Field 1 (CP/M exit) —This is one of several holdovers from the venerable CP/M operating system. Any program which wishes to exit can do so by jumping to offset 0 of the PSP. This is *not* a recommended way for a program to exit; use function 0x4C of INT 0x21 instead.

Field 2 (*Memory size*) —This number represents the number of paragraphs (16-byte chunks) of memory available to the program. It can be handy to check this number if you have a memory-intensive program, just to be sure you have enough room.

Field 4 (Function dispatcher call) —A FAR CALL to the MS-DOS function dispatcher, normally accessed through

a software interrupt (INT 0x21).

Fields 5 through 7 (Saved interrupt vectors) —MS-DOS stores the values of these important interrupts when a program starts up, and then restores them from this area when the program exits. This is insurance in case the program modifies these vectors without restoring them later.

Field 8 (Parent PSP segment) —Here we find the PSP segment of the parent program, which is normally COMMAND.COM.

Field 9 (Local file handle tabe) —If you've ever wondered why a program is limited to only 20 open files, this area explains it. This table contains 20 1-byte entries, each of which is an index into the internal MS-DOS file handle table. A value of 0xFF in the local table indicates that that file has not been opened. The first five entries are assigned the default standard input, standard output, standard error, standard aux, and standard printer handles. This explains how MS-DOS does I/O redirection —by placing the "internal" handle of a file in, say, standard input, rather than the handle which represents the CRT. Note that you can (carefully) change these values, doing your own I/O redirection from inside your program.

Field 10 (Local environment segment) —This word is used to find the local copy of the environment, which MS-DOS creates when starting the program.

Field 11 (Local stack storage) —When an MS-DOS interrupt is invoked, the stack segment and pointer of the program are stored here, while MS-DOS switches to an internal stack. When the interrupt is done, the program's stack information is restored.

Field 12 (File handle table size) —Contains the total number of entries of the local file handle table.

Field 13 (Address of local file handle table) —Changing this value seems to have no effect in MS-DOS prior to Version 3.30, which has a documented function allowing you to expand the size of the local file handle table. That function (0x67) allocates a new memory block (to hold the new table) via function 0x48, and, if successful, stores the address of that block here.

Field 15 (MS-DOS function call) —Yet another way of getting to the MS-DOS function dispatcher, these three bytes contain the instructions INT 0x21 and FAR RET.

Fields 17 and 18 (Default file control blocks [FCBs]) —These are remnants of CP/M compatibility, and are virtually useless. The first two command line parameters are parsed into this area under the (usually false) assumption that they are file names. Apparently the undocumented areas on either side of these FCBs are used by the extended versions of FCBs. Starting with MS-DOS, Version 2.x, the only real use of FCBs is to access special files, such as a volume label.

Field 20 (Command tail length) —Contains the length of the command tail.

Field 21 (Command tail) —The command tail is a modified version of the command line used to start the program. It begins with the first space after the program name and contains everything up to and including the carriage return. Any I/O redirection operations (for example, > *output.dat*) are removed. It is terminated by a null (0 byte). Although many programming languages (most notably C) automatically parse the command line for you, it is sometimes necessary to parse the command tail yourself for special purposes.

TSR Functions

Everyone these days seems to be writing terminate-and-stay resident (TSR) utilities. These programs range from simple time-display utilities up to system tyrants, such as SideKick Plus. I've written several myself, and have found that well-behaved TSRs *must* use undocumented MS-DOS features.

A well-behaved TSR is one which peacefully and cooperatively co-exists with MS-DOS, other TSRs, and regular applications. Unfortunately, Microsoft has never published the complete details of how to go about writing such a program.

Many TSRs are designed to be "pop-up" utilities, which open a processing window whenever a special "hot key" is pressed. Such programs provide a limited form of multiprocessing, wherein the user can call up a TSR function (such as a spelling checker or notepad) from within any application. But MS-DOS is not reentrant; that is, it cannot be interrupted while performing a critical operation such as writing to the disk. Because a TSR is activated by an asynchronous event (such as a timer tick or a keystroke), it is vital for the TSR to know when it can't interrupt MS-DOS.

MS-DOS maintains a special undocumented variable known popularly as the INDOS flag. When MS-DOS is doing something critical, it increments this one-word flag. Thus MS-DOS can be safely interrupted when the flag is zero. The location of this flag can be found through function 0x34, which has these particulars:

Function 0x34: Get INDOS Flag Address
Interrupt: 0x21
Function: 0x34
Registers on entry:

Field No.	Byte Offset	Byte Length	Field Name
1	0x00	2	CP/M exit
2	0x02	2	Memory size
3	0x04	1	Unknown
4	0x05	5	Function dispatcher call
5	0x0A	4	Saved DOS terminate vector
6	0x0E	4	Saved CTRL-BREAK vector
7	0x12	4	Saved critical error vector
8	0x16	2	Parent PSP segment (undocumented)
9	0x18	20	Local file handle table (undocumented)
10	0x2C	2	Local environment segment
11	0x2E	4	Local stack storage (undocumented)
12	0x32	2	File handle table size (undocumented)
13	0x34	4	Segment/offset of local file handle table (undocumented)
14	0x38	24	Unknown
15	0x50	3	MS-DOS function call
16	0x53	10	Unknown
17	0x5C	16	Default file control block 1
18	0x6C	16	Default file control block 2
19	0x7C	4	Unknown
20	0x80	1	Command tail length
21	0x81	127	Command tail

Table 1: *The PSP format*

AH = 0x34
Registers upon return:
ES = segment of INDOS flag
BX = offset of INDOS flag

Using EX:BX as a pointer, a program can check that it is safe to interrupt MS-DOS. Not only is this useful for TSRs, but it is also necessary for multitasking programs.

One complexity with using the INDOS flag is that it is set to a value of 1 while MS-DOS is waiting at the command prompt. Obviously MS-DOS isn't *really* doing anything—it's just waiting for input. Luckily we have another undocumented facility at our disposal: the idle interrupt.

Whenever MS-DOS gets bored and has nothing else to do, it invokes an INT 0x28. Programs can chain to this interrupt, and know when they receive it that MS-DOS is not doing anything important. When I write a TSR that is activated from the keyboard, I capture both the hardware keyboard interrupt (INT 0x09) and INT 0x28. When I get the INT 0x09, I examine the INDOS flag to see if DOS is busy before I do anything. Whenever I see an INT 0x28, I ignore the INDOS flag and just look to see if the hot key has been pressed.

MS-DOS tracks (internally) pieces of information about the currently active program. When a TSR is activated, it is necessary to change MS-DOS's information so that the TSR is not using the resources of the currently active application.

One resource of particular interest is the program segment prefix, discussed earlier. MS-DOS assumes that the program most recently loaded is the currently active program and stores this PSP in an internal variable. For example, when you open a file, MS-DOS uses the handle table in the PSP of the most recently executed program.

This can cause some subtle problems when using TSRs. When a TSR is activated, we need to save the current PSP, and tell MS-DOS to use the PSP of the TSR. Otherwise, when the TSR uses a PSP resource (such as in opening a file), it could corrupt the PSP of the program most recently loaded. This could fill up the most recent application's file handle table or crash its stack.

There are ways around this problem. MS-DOS function 0x50 sets the current PSP, and functions 0x51 and 0x62 retrieve the PSP of the current program. These functions are accessed as follows:

Function 0x50: Set Current PSP
Interrupt: 0x21
Function: 0x50
Registers on entry:
AH = 0x50
BX = Segment of the PSP to be made current
Registers on return:
None

Functions 0x51 and 0x62: Get PSP Segment

Interrupt: 0x21
Function: 0x51 (undocumented, MS-DOS 2.x and later)
0x62 (documented, MS-DOS 3.x and later)
Registers on entry:
AX = 0x51
Registers on return:
BX = Current PSP segment

One of the first acts of a TSR during its initialization should be to retrieve and store internally the segment of its own PSP. Some high-level languages (for example, Turbo Pascal and most C compilers) have a global variable which is automatically set to the segment of the PSP. Either way, the segment should be retrieved before the TSR becomes resident.

Why are there two functions used to retrieve the current PSP? Well, function 0x51 existed beginning with MS-DOS 2.0 and had a serious bug when used with an INT 0x28 handler (it did not restore the stack properly). Function 0x62 was introduced with MS-DOS 3.0 and does not have the bug. Why Microsoft didn't just fix the bug with 0x51, rather than adding a new function, is anybody's guess.

Conclusion

I've covered some of the more useful and interesting undocumented (or poorly documented) functions and features of MS-DOS. The PSP is an integral part of a program, and proper use of its resources can enhance and extend the performance of a program.

As for the TSR information I presented earlier, it seems odd that Microsoft would document how to make a program resident, but not how to make it perform in a well-behaved manner! If you're writing a TSR, you can use this information to avoid the numerous pitfalls involved in designing memory-resident programs.

Bibliography

MS-DOS 2.1 Programmer's Reference. Redmond, Wash.: Microsoft, 1983.

Duncan, Ray. *Advanced MS-DOS.* Redmond, Wash.: Microsoft Press, 1986.

Ladd, Scott Robert. "A Turbo TSR." *BYTE* 13(7) (July 1988): 301 – 304.

Young, Michael J. *Performance Programming Under MS-DOS.* Alameda, Calif.: Sybex, 1987.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

EGA and VGA
Smooth Scrolling and Panning

Smoother screen transitions using video controller registers

by Andrew J. Chalk

Although PC applications have traditionally relied on BIOS-type scrolling, smooth scrolling will figure more prominently in future applications. User expectations for WYSIWYG displays placed increased demands that programs offer oversized virtual screens and deftly handle rapid transitions of document/screen perspective. In turn, the proliferation of high-resolution monitors driven by EGA and VGA cards has made programs that draw upon the expanded features of these cards and monitors a more worthwhile use of development time.

Smooth scrolling and panning take advantage of the EGA and VGA adapters' ability to move the viewing window under the display one pixel at a time in any direction. This can be done in both text and graphics modes by manipulating various EGA/VGA video controller registers in ways that weren't possible with the older CGAs and MDAs.

This article describes a Microsoft C-compatible library that allows you to integrate text-mode smooth scrolling and panning into your applications. Smooth scrolling works by updating the start address by one or more scan lines at a time inside a consistent timing loop, using the built-in hardware support of the EGA or VGA card. The result is that the entire display appears to slide up, down, left, or right, with smooth movement at a controlled rate. This process uncovers portions of the display that were previously off-screen. The effect is almost magical.

The code presented here was tested on a number of EGA and VGA cards (including the VGA on the PS/2). However, the register-level compatibility of certain EGA and VGA clones with their IBM counterparts is less than complete. With third-party VGAs in particular, you should try your code on a PS/2 if you experience problems.

A file lister (or browser) called *smooth browse* is also included here. It lets you load a text file and move through it, with smooth scrolling and panning, by means of the cursor keys. The source code for *smooth browse* is compatible with Microsoft C, Borland Turbo C, Watcom C, and Mix Power C.

To develop a library that's flexible enough to handle either an EGA or a VGA at run time, you must be sensitive to the similarities of and differences between the two adapters. Fortunately, the VGA and its predecessor, the EGA, are more compatible than the EGA and the CGA. However, since smooth scrolling and panning are not supported at the BIOS level, there are important differences that the code must account for. In this article, the term "the adapter" will be used with the information is true for both adapters.

While the following discussion applies to text-mode applications, much of it is also relevant to graphics mode. In an ironic twist of the usual state of affairs, smooth scrolling and panning are more difficult to implement in text mode than in graphics mode.

How Smooth Scrolling and Panning Work

The EGA is configured with up to 256K of read/write memory in multiples of 64K. Although IBM's original EGA came with only 64K as its standard, clones almost invariably are fully populated with RAM. IBM released the VGA with 256K as standard. The size of the display buffer is 32K on VGAs and EGAs that are fully populated with 256K of RAM. EGAs with less video memory have proportionately smaller video buffers but are comparatively rare. In recognition of this fact, we will assume that your adapter has 256K.

To ensure compatibility with earlier adapters, the address of the display buffer in the processor address space depends on the active video mode. Video modes 0–6 are CGA compatible and map into segment B800H of the processor address space. Mode 7, the monochrome mode, maps into segment B000H of the processor address space. All other video modes are EGA- and VGA-specific; they map to segment A000H.

Although a character and attribute

Andrew Chalk is the president of Magna Carta Software, which offers the C Windows Toolkit, a video function library containing extensive support for the EGA and VGA. He can be reached at P.O. Box 475594, Garland, TX 75047.

may have been written to display memory through DOS calls, the PC BIOS, or by direct memory mapping, it is easiest to understand smooth scrolling by assuming that our program writes directly to the display buffer. In this way we can think of the screen as an array of memory addresses, without confusing the subject with DOS or BIOS calls (which ultimately write to the screen as an array in memory addresses anyway). IBM calls the memory address that appears at the top-left corner of the screen the "start address." Whenever our program writes data to the start address and subsequent addresses within the display buffer, the data can be displayed on the CRT.

For example, if our program uses 80-column color text mode, the character written to the top-left corner of the screen is at address B800:0000H. Since the attributes of a text-mode character are determined from the byte at the next memory address (called the "attribute byte"), the attribute for this character is determined by the contents of the byte at B800:0001H.

If each screen row has 80 characters, each of which is followed by an attribute byte, then a character row requires 160 bytes of storage. In 25-line mode this implies that a single screen is 4,000 bytes of display memory. Note that this is just under 4K of memory on an adapter that contains far more video memory. If we write to the display buffer at addresses that are greater than those on the screen, our text is not immediately visible, but it can be in the display buffer nonetheless. The usual way to display it is through BIOS INT 10H, service 5 (set active display page).

On the EGA and VGA, register AL can be loaded with any value from 0 to 7 to select one of eight video pages. A video page is the 4,000 bytes of memory we have already accounted for in the example just given, plus a 96-byte margin between pages. Thus, interrupt 10H, service 5, permits us to move through the display buffer in 4,096-byte chunks. As we select successively higher page numbers, the BIOS actually shows us higher display-buffer addresses, and any data written to those addresses becomes visible.

Smooth Scrolling

Smooth scrolling takes the principle behind the BIOS video paging service to its logical conclusion. The basic idea behind smooth scrolling is to update the start address by one or more scan lines at a time inside a consistent timing loop. To set the value of the start address (as a byte offset from the beginning of the display buffer), you program the desired value into the CRT controller start address low and start address high registers. For example, to set the start address to 1,000H, write 10H into the start address high register and 00H into the start address low register. In the source code shown in Listing One, page 101, setting the start address is handled by the function *set_start_addr( )*.

If we update the start address by one screen width (normally 80 columns) each time through the loop, the text on the screen moves up by a whole character row. To create increments

Smooth scrolling takes the principle behind the BIOS video paging service to its logical conclusion

the size of one scan line, we use the preset row scan register in the CRT controller. This register has a default value of 0. Each time that we increase it by 1, text on the screen moves up one scan line. There are 14 scan lines per character row (by default) on the EGA attached to an enhanced color display or a monochrome display, and 16 scan lines on the VGA. There are only 8 scan lines per character row on a CGA monitor (or an EGA in 43-line mode).

Smooth scrolling is handled by the function *smooth_scroll( )*. Before this function is called for the first time, the global variable *vpel*, short for vertical pel value, is 0 (a pel is IBM parlance for a pixel). The algorithm used in *smooth_scroll( )* increments the value of *vpel* in a loop until the screen character height is reached. Then the start address is incremented by the length of one screen width, and the pel scrolling begins anew with *vpel* once again 0.

Consistent timing is important when you smooth scroll, or the text will move at varying speeds. For this reason we must not try to control the timing through a loop in our program (which would be dependent on processor speed). Instead, we use built-in adapter hardware support. The image on the enhanced color display is updated 60 times a second. After each update, the CRT gun moves back to its starting position (the top-left corner of the CRT) in an action known as a "vertical retrace." Because this 60Hz refresh cycle is independent of CPU speed, if we can detect the start of the vertical retrace we can use it as a signal to our program to update the display-buffer start address and achieve our desired slow loop.

Fortunately, both the EGA and the VGA provide support to detect the vertical retrace that occurs at a processor-independent speed. To poll for the start of each vertical retrace before updating the start address, we read the input status register and AND the value with 8. As many readers are aware, the EGA and PS/2 VGA can be programmed to generate an interrupt on IRQ2 at the start of each vertical retrace. We choose polling instead because the interrupt is not supported on the VGA display adapter (the card for the original IBM PC AT), and it is not certain that VGA clone developers will choose to implement it.

Smooth Panning

The principles behind smooth panning are similar to those of smooth scrooling, although the implementation is more involved. Suppose that we update the display-buffer start address by two bytes. In this case the first character on the screen disappears, and all other characters march to the left. Text moves sideways, but not smoothly. This is not panning, because the character in the first column of the second line moves to the end of the first line and so on. We want all screen lines to move to the left one character so that the leftmost character of every row disappears. This is easily achieved with the EGA and VGA due to built-in support for logical line lengths. Each CRT line can be wider than the physical screen (up to 512 bytes). If we redefine the logical line length to more than 80 characters (by addressing the CRT controller offset register at index 13H), the adapter obligingly moves all lines to the left.

Our screen update is not smooth, because we have advanced the start address by two bytes (one character and its attribute), which is too large an amount. To achieve smoothness we use two things. First, the CRT vertical retrace is used as it is in smooth scrolling to achieve consistent timing. Second, the EGA and VGA provide a register in the attribute controller at index 13H, called the horizontal pel panning register, that allows pixel-sized increments in the position of each character,

permitting the appearance of smooth movement.

In the source code for smooth browse (Listing One), the program performs smooth panning by first setting a logical screen width larger than 80 characters and then executing two loops. The outer loop updates the screen start address and the inner one increments the horizontal pel panning register. The net effect is that the user sees a smoothly panned screen.

A Dissection of the Source Code Library

The code for the smooth scrolling and panning library, Listing Two, page 108, illustrates one way to implement smooth scrolling and panning in text mode on the EGA and VGA. The library requires knowledge of certain system variables, such as whether the system contains an EGA or VGA, the active video mode, the character height, and so on. Source code for functions that perform these supporting roles is included in smooth browse. The functions that our program must perform prior to calling the smooth scrolling and panning library are: First, determining that an EGA or a VGA is present and active. Second, set the video mode to the one desired (if the system is not already in the correct mode) and obtain the number of screen columns. Third, detect the type of monitor installed, the amount of video RAM on the adapter (if it is an EGA), and the character height. The addresses of the input status register and the CRT controller index register depend on the presence of a monochrome or color monitor and can be derived once the type of monitor is known. Finally, the cursor size should be saved for restoration on exit.

Before it calls any of the library functions, the calling program should also perform some basic setup tasks. First, the variable *left_edge* must be initialized to TRUE, indicating that we are at the left edge of the file to be displayed. Second, the variable *end_of_buffer* must be calculated as the size of the file that is loaded divided by 2 (to allow for attribute bytes). Third, and far from obvious, the input status register must be read to initialize the attribute controller.

There are no special rules regarding the order in which you call the functions in the library, save that if you want to use a logical screen width that is different from the default (80 in video mode 2 and video mode 3), you should call *set_logical_screen_width( )* before calling any of the other functions. A word is in order about the arguments to *smooth_scroll( )* and *smooth_pan( )*. The first argument, *count*, is the number of scan lines to be scrolled or panned (negative values move back to the top and left of the video buffer). This value may be low (equal to just 1 or 2) in each call to these functions, or it could be several hundred. Although the latter implies less overhead between calls to the loops within these functions that actually do the scrolling, my experience has been that even with frequent calls to *smooth_scroll( )* and *smooth_browse( )* that scroll only one or two scan lines at a time, the overhead does not produce any discernible difference in speed for the user. Smooth browse illustrates this point.

The second argument in *smooth_scroll( )* and *smooth_pan( )*, which is *speed*, is the desired speed of movement. This variable adjusts the speed by dictating the number of scan lines by which the screen is scrolled in each loop by *smooth_scroll( )* and the number of pels by which it is shifted sideways by *smooth_pan( )*. If *speed* is set to 1, the screen slides slowly and eerily in the designated direction. If it is set to 5, the screen glides past the user at a rate that is too fast to read.

The function *smooth_scroll( )* implements the smooth scrolling techniques described previously. The first thing that *smooth_scroll( )* does is read the start address from the CRT controller registers. The start address is stored in two registers, start_address_high (0CH) and start_address_low (0DH). These registers hold the high byte and low byte, respectively, of the screen start address. We OR the two register values together, after shifting the high byte left 8 bits, and assign the value to *screen.start_addr*.

The function *smooth_scroll( )* has two loops to perform. For each character row, the start address must be incremented by the number of bytes in a logical screen line in order to move us to the next character line. Within each character line, the start address is fixed, but the starting scan line of the first character row on the screen is incremented from 0 to the character height by incrementing the value in the preset row scan register (index 8 of the CRT controller).

The function *smooth_pan( )* is similar in operation to *smooth_scroll( )*. Given a panning direction, there are two loops. One of these pans a full character by incrementing the start address by 1. The other writes the value of *hpel* (horizontal pel panning) to the horizontal pel panning register of the attribute controller chip and increments *hpel* by the value of *speed* for each

iteration. *smooth_pan( )* looks more complicated than it really is. This is due to the different pixel width of the EGA monochrome and VGA characters (9 pixels) versus EGA color characters (8 pixels). The code has to perform an extra conditional test to see if video mode 7 or a VGA is active so that the value of *hpel* can be iterated through 8,1,2,3,4,5,6,7,0, where 8 is the default setting at which video memory is addressed on a byte boundary. On the color EGA, the valid values for *hpel* are 0,1,2,3,4,5,6,7, with 0 being the default setting. Contrary to expectations, smooth panning is easier in graphics modes. With the exception of VGA mode 13H, *hpel* would be incremented through from 0 through 7.

Consistent timing is important for smooth scrolling or when text will move at varying speeds

The other thing worth noting about *smooth_pan( )* is that it uses the defined logical screen width to set two values, *left_edge* and *right_edge*, so that panning respects the logical line width. After panning one logical line width to the right, panning stops. The screen can be panned back to the left, and it stops at column 1.

What do you do at the end of a smooth scrolling and panning session? Many registers on the adapter may not be in a desirable state and so must be reset. Although you could write to each one separately, you may wish to take the Chuck Norris approach to video control and just reset the video mode before exiting your program.

Limitations and Enhancements

In the earlier description of smooth scrolling and panning concepts, I glossed over two important hardware limitations. The first is the limited display-buffer size. By default this is 32K, but

SCROLLING
(continued from page 66)

it can be increased to 64K fairly easily (smooth browse shows how). One way around this limitation (and one that could be introduced into smooth browse) is to use dynamic display-buffer allocation. This could be implemented by storing the object in RAM in the transient program area and mapping it into the display-buffer on cue.

The second limitation is the imposition that the whole screen must be scrolled or panned. The EGA and VGA can produce a split screen in which one screen is kept stationary while the other is scrolled (also used in smooth browse), but this is only a minor amelioration of a major hardware limitation. Some EGA/VGA clone makers have responded to this situation by building in hardware support for smooth-scrolling windows, but we will have to wait for the next generation of video adapters until such features are generally available to support software windowing.

Summary

The main purpose of this article was to explain the useful programming technique of smooth scrolling and panning on a machine equipped with an EGA or a VGA. It illustrated how to implement this technique with a smooth scrolling and panning library. There is also an example application, smooth browse, that uses these techniques, plus the hardware split screen and 16 video pages that are available to *DDJ* readers. The example applied these techniques in text mode, but they also work (with some small amendment to the sample source code) in graphics mode. In fact, the graphics mode implementation is generally easier than the text mode implementation.

Once you see smooth panning and scrolling in action, new possibilities suggest themselves, and I am sure that *Dr. Dobb's* readers will find many new applications in which to incorporate them.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal,* 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 101).

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

ARTICLES

Find That Function—From Inside Brief!

Here's a Brief macro that locates function definitions without exiting Brief

by Marvin Hymowech

For those readers who appreciated the convenience of my function finder ("Find That Function!" *DDJ,* August 1988), this article presents a useful refinement: A Brief macro that prompts the user for the name of a C function to be located, then presents the source file containing the function definition in the current editing window. With this macro, successive function calls can be traced without exiting Brief and without ever having to remember which source file contains which function.

getf Becomes a Macro

Recall my previous function finder, and you'll remember that it consists of the two programs, *bldfuncs* and *getf.* The first program, *bldfuncs,* reads C source files, extracts the names of functions defined therein, and constructs an index file (named *funcs.txt*) of source files and functions, in the form:

```
file_1.c:
    function_1a
    function_1b
    . . .
    function_1z;
file_2.c:
    function_2a
    . . .
```

The second program, *getf,* then searches *funcs.txt* for a specified function name, extracts the relevant C source file name,

Marvin Hymowech works as a programmer for Symplex Communications Corp. Previously he taught mathematics at the University of Michigan in Ann Arbor. He can be reached at 4906 Cole Blvd., Ypsilanti, MI 48197.

and replaces itself in memory with a text editor using this source file as the file to be edited.

Although *getf* works well, it can only be invoked from the DOS level. A valuable addition to this mechanism would be a way to use something like *getf* after you are already in Brief, without having to exit to DOS first. There is an obvious way for the user to continue finding functions from inside Brief, via the following steps:

1. Open a new window containing *funcs.txt*
2. Search for a function name
3. Search backwards for a : (colon)
4. Read the name of the relevant source file, now on the current line
5. Edit this source file in a new window
6. Search for the function name in this source file until you find it

The Brief macro language contains all of the capabilities required to automate this process. (I use Brief, Version 2.01.) Listing One, page 72, contains the Brief macro *getf* (it is the macro analog of the program *getf*) as well as a slightly improved version of the macro *funcsrch* (which was presented in my original article). To maximize ease of use, *getf* should be bound to the Ctrl-F key by placing the following lines

```
(autoload "getf.cm" "getf" "funcsrch")
(assign_to_key "<Ctrl-f>" "getf")
```

in the *keyboard.h* file. (This file is included in the macro startup, which runs automatically when Brief is first invoked.) Of course, you must first use Brief to compile both *getf.m* and *startup.m*, which produces *.cm* files (see the Brief documentation for the details).

How getf Works from Within Brief

Let's look at the macro *getf* in detail. First, let's get a function name from the user using the Brief function *get_parm*:

```
( get_parm NULL fn_name "Enter
  function name: " NULL fn_name ) )
```

Here, the first NULL parameter instructs *get_parm* to prompt the user (*get_parm* is also used to get parameters passed by other macros) by displaying "Enter function name:" on the bottom message line of Brief. The string the user types is then placed in the variable *fn_name*, which is declared as a global string variable so that it will retain its value between calls to *getf*; this allows us to present its last value as the default, and display it after the prompt string.

Next, check that the file *funcs.txt* exists in the current directory; if not, the message "file *funcs.txt* not found" is printed on the message line and the macro returns. Assuming *funcs.txt* is found, create a buffer for it using *create_buffer*, then make it the current buffer using *set_buffer*. Note that the user will not see any change on the screen as yet, because this buffer has not been attached to a window.

Now *funcs.txt* can be searched for the function name. Care must be exercised, however, because simply finding a string matching the function name is not enough: This might be a part of a larger function name or might occur as the name of a source file. After finding an occurrence of the string, the line is read into the string variable *line* and is examined as follows: the line must have a tab (*\t*) as its first character in order to be a line containing a function name (source file names are not indented in the standard format of *funcs.txt*), and the line must consist exactly of a tab, the string contained in *fn_name*, and a newline (*\n*). Note that a semicolon is allowed before the newline —this would occur if the function were the last entry for a source file. If the line matches these specifications, a search backwards from this point for the first : (colon) is performed to get the name of the source file; otherwise we'll advance to the next line and search for the next occurrence of the function name. If at any point the function name is not found, "function not found" is displayed on the message line and the macro returns.

Having obtained the line containing the source file name, this name is extracted and placed in the string variable *file_name*. After first verifying that this file exists in the current directory, it is read into a buffer and attached to a window using the single Brief function *edit_file*. Control is now transferred to the macro *funcsrch* (explained in my previous article) in an attempt to position the cursor on the exact line of the function definition.

With this macro, you can trace successive function calls without exiting Brief and without ever having to remember which source file contains which function

After *getf* is executed, the file *funcs.txt* remains in a buffer during the editing session, which speeds up subsequent calls to *getf* since *funcs.txt* need not be reloaded each time.

Using Wildcards?

The program *getf* (in my previous article) has a capability that this Brief macro-version of *getf* lacks: the program version was able to search for function names based on a mask containing wildcard characters, and then present the user with a menu of matching names. It should be possible to add this feature to the macro *getf* by using Brief's Dialog Manager to construct a system buffer of available choices and to present this menu to the user. I would be interested in hearing from any reader who pursues this.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 72.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Listing One *(Text begins on page 70.)*

```
;**   getf - Copyright 1988 by Marvin Hymowech.
;**        - Prompts for a function name, extracts the name of the file
;**        - in which it is defined from "funcs.txt" (see bldfuncs.c)
;**        - and presents the file in the current window.
;**        - Returns: 0 if error, else 1.

#define NON_SYSTEM 0

( macro getf
   (
      ( int start_buf_id func_buf_id )
      ( string fn_name line file_name )

         ;** make fn_name a global so that it will retain its prior
         ;** value for the default display in the get_parm below
      ( global fn_name )


         ;** get desired function name from user
      ( if
         ( ( !
            ( get_parm NULL fn_name "Enter function name: " NULL fn_name ) )
         )
         (return 0)
      )

         ;** verify that funcs.txt exists on disk in the current directory
      ( if ( ! ( exist "funcs.txt" ) )
         (
            ( beep )
            ( error "file funcs.txt not found" )
            ( return 0 )
         )
      )

         ;** save current buf id, so we can restore it if an error occurs
      ( = start_buf_id ( inq_buffer ) )

         ;** create a buffer for funcs.txt without attaching it to a window
      ( if
         ( ! ( = func_buf_id
               ( create_buffer "functions" "funcs.txt" NON_SYSTEM )
            )
         )
         (
            ( beep )
            ( error "Can't load funcs.txt file." )
            ( return 0 )
         )
      )

         ;** make it the current buffer
      ( set_buffer func_buf_id )

        ;** Brief back-end buffer functions won't set tabs for us
         ;** like edit_file, so we do it explicitly
      ( tabs 9 17 )

      ( top_of_buffer )

         ;** search for the function name
      ( while 1
         (
               ;** break if function name not found
            ( if ( <= ( search_fwd fn_name ) 0 )
               (
                  ( beep )
                  ( error "function %s not found" fn_name )
                     ;** restore the original buffer
                  ( set_buffer start_buf_id )
                  ( return 0 )
               )
            )
               ;** get the current line into variable "line"
            ( beginning_of_line )
            ( sprintf line "%s" ( read ) )

               ;** if fn line, not file line
            ( if ( == ( index line "\t" ) 1 )
               (
                     ;** read the line into "line" starting after the tab
                  ( next_char )
                  ( sprintf line "%s" ( read ) )

                     ;** make sure we have the exact function name, not
                     ;** part of a larger function name

                     ;** if the line consists of \t,fn_name, and \n,
                     ;**    then break
                  ( if ( == ( index line "\n" ) ( + 1 ( strlen fn_name) ) )
                        ( break )
                  )

                     ;** if the line consists of \t,fn_name, ;, and \n
                     ;**    then break
                  ( if ( == ( index line ";" ) ( + 1 ( strlen fn_name ) ) )
                        ( break )
                  )
               )
               ( end_of_line )               ;** else try next line
            )
         )

         ( if ( <= ( search_back ":" ) 0 )   ;** done, get the file name
            (
               ( beep )
               ( error "funcs.txt corrupted" )
               ( set_buffer start_buf_id )
               ( return 0 )
            )
         )

         ( beginning_of_line )
         ( sprintf line "%s" ( read ) )
         ( sprintf file_name "%s"            ;** get rid of trailing ":"
            ( substr line 1 ( - (index line ":" ) 1 ) ) )


            ;** verify that the file exists on disk
         ( if ( ! ( exist file_name ) )
            (
               ( beep )
               ( error "file %s not found" file_name )
               ( set_buffer start_buf_id )
               ( return 0 )
            )
         )

            ;** and edit it
         ( edit_file file_name )
            ;** try to place cursor on the function defn
         ( return ( funcsrch fn_name ) )
   )
)

;** funcsrch - Copyright 1988 by Marvin Hymowech.
;**          - Searches the current buffer for a string (function name)
;**          - by starting from the end of the buffer.
;**          - This is designed to position the cursor on
;**          - a function definition since most functions tend to be
;**          - used before they are defined.
;**          - Returns: 0 if function not found, 1 if found.
(macro funcsrch
   (
      ( string s )
      ( extern _s_pat _dir center_line )

      ( get_parm 0 s )
      ( message "locating function %s..." s )
      ( end_of_buffer )
      ( if ( > (search_back s) 0 )
         (
            ( message "function %s found" s )
            ( center_line )
            ( sprintf _s_pat "%s" s )
            ( = _dir 0 )
            ( return 1 )
         )
      ;else
         (
            ( top_of_buffer )
            ( beep )
            ( error "function %s not found" s )
            ( return 0 )
         )
      )
   )
)
```

End Listing

A Class-ier C

C_talk, where object-oriented programming meets C

by Ernest R. Tello

C_talk from CNS (Eden Prairie, MN) is an object-oriented hybrid of the C and Smalltalk programming languages. It couples the essential features of an object-oriented language such as inheritance, encapsulation, message passing, and run-time binding to the speed and efficiency of the C language.

In Prolog/V: "Prolog in the Smalltalk Environment" (see November 1988 issue), Gregory Lazerev took a detailed look at the synergy that developed from fusing the strengths of Prolog/V and Smalltalk/V into a potent development environment. This article explores how coupling the procedural language C to a Smalltalk-like object-oriented software environment can yield equally impressive dividends by allowing the incremental and interactive development of an object-oriented system written in familiar, portable C.

The C_talk system consists of three main programs: a Smalltalk-like browser, a preprocessor compiler, and a Make utility. The C compilers currently supported are: Microsoft, Lattice, Turbo, and C86. It's noteworthy that the same system supports all of these compilers. So no additional cost is incurred if you choose to switch compilers in midstream.

The feat that deserves the most attention is CNS's ingenious way of incorporating the Smalltalk-like browser into the C programming environment. The browser forms the real center of this programming tool. It can be used interactively just like the Smalltalk browser (in spite of the fact that C is a compiled rather than an interpreted language). To get an idea of the advantages presented by programming with a tool like C_talk, let's take a look at this innovative browser.

Ernie Tello is a consultant, lecturer, and writer in the field of artificial intelligence. He can be reached at 1518 W. Cliff Dr., Santa Cruz, CA 95060.

The C_talk Browser

It's surprising how similar the C_talk browser is to the Smalltalk facility that it was modeled after. Like its parent, the C_talk browser acts as a combination editor and application manager that allows object-oriented systems to be developed incrementally and interactively. That's no mean achievement for a C-based object-oriented tool.

The browser exists as a stand-alone program that's called like any stand-alone application. As it will be seen, however, it's this program that provides an extremely powerful facility for integrating virtually the entire C_talk system.

Not only does the program allow you to familiarize yourself with the environment through interactive browsing, but it can also be used for doing many of the programming chores interactively, including creating a Make file for batch processing the compilation and linking operations.

The C_talk browser is divided into a number of panes that operate like tiled windows, in that these apertures cannot be overlapped or re-sized. The upper left pane displays the class hierarchy. Just as with a Smalltalk browser, when a class is selected, its methods are shown in the upper right window pane. Then a method can be selected and its code is made available for editing in the bottom text pane.

Alternatively, any text file can be loaded into the browser and displayed in the lower text pane to be edited. Thus, the browser fills the dual purpose of a small, simple editor and a tool specially equipped for handling C_talk objects.

To operate the browser, various pop-up menus are available in different pane areas. These pop-ups can be activated either by mouse or key commands. Included in the browser facilities is a shell that allows you to issue commands to the operating system from within the browser or to temporarily exit to the operating system for doing various chores, and return to the browser just as you left it. This kind of shell has become so widespread that it is now really conspicuous when it is absent.

There is an important difference between the C_talk browser and a Smalltalk browser. The former is a stand-alone application in a compiled environment and the latter is part of an interactive system in which everything is omnipresent. When the C_talk browser is first loaded, (because it does not have the classes as part of itself) it looks for all the class definitions that exist in the current directory on disk and compiles them into its database so that it can provide interactive access to them.

Once everything has been loaded into the browser, however, many functions needed for writing applications can be done interactively, so that the browser actually generates the code. Such functions include specifying a new application, loading existing classes, defining new classes and their methods, and saving all work. There is also a zoom function that expands to the size pane of a full screen.

Everything was not yet "ideal" in the version of the browser I tested. For example, the ability to load text files into the editor is confined to rather small files. If the file you load into the browser editor is larger than the maximum size, only the amount that fits will be loaded, and no error message is given. This can be dangerous for the unwary, because if you were to make a small change and save such a file this could result in a substantial loss of data. For this reason, I would say that

the browser is useful primarily for the "build" facilities that allow you to create various small files that will form modules in a complete system. For editing relatively large files I recommend that another editor be used.

C_talk Syntax

The syntax of C_talk is slightly different from other object-oriented C dialects, but if you already know C, once a few conventions are mastered it becomes relatively straightforward to read and write. The main convention concerning C_talk messages is that they are always flanked by two "at" (@) symbols on either side. So, for example, if we want to create a new instance of a class such as Rectangle, then the message would look like this:

```
@Rectangle new_ &rect@;
```

Here the ampersand character is used for pointer notation exactly as in C.

The *id* statement is a C_talk class id declaration. In C_talk programs you must make external declarations for each class id that is referenced.

One of the most powerful features of C_talk is the ability to map one message to another. This means that the selector, or name of a message, can be passed as an argument to another message. The syntax for doing this is a variable assignment statement with the name of the message selector enclosed by backquote characters.

To actually send a message that references another message that has been stored as a variable, special messages are provided as methods of the Object class that are inherited by all other objects. These methods are *perform_*, *perform_with_*, and *perform_with_with_*.

So, for example, if we have defined the variable *gval* as follows:

```
id gval;
gval = 'getValue';
```

and an object,

```
id obj;
int gval;
```

Then the *getValue* message could be mapped by the expression:

```
@obj perform_ gval with &val@;
```

Here it is assumed that *val* is a slot in *obj*, and that the *getValue* message takes this as an argument.

As with most object-oriented languages, the pseudo variables *self* and *super* are used to reference objects within a message. The *self* pseudo variable references the object to which the message is sent. The *super* pseudo variable references the superclass of a class. To access the instance variable of the object to which a message is sent, the following notation is used:

```
self-> width
```

assuming that *width* is the name of an instance variable for the object in question.

Another C_talk syntax convention involves the use of the underscore character (_). Whenever an underscore follows a message selector name, this indicates that an argument is to follow. From a practical point of view, one of the more important things about an object-oriented C system is the size of its built-in foundation classes, so we'll look at this next.

C_talk Foundation Classes

The Container class is an abstract or formal class that is used for creating subclasses that provide dynamic data storage elements. (See Table 1, page 77.) This means that a subclass of Container can allow instances to be created that have either fixed- or variable-sized data elements, as well as sequential or non-sequential access methods. So, for example, if the *expand* message is sent to an object, the object is doubled in size.

For more precise size expansion, the *expandBy_* method is used, which takes an argument that specifies the number of additional data elements the expanded object will contain. The *putRecSize_* method similarly sets the size in bytes for each data element that the object contains. To access an indexed object, you use the *at_ get_nRecs_* method, which returns as many records as you specify, starting at the specified index number.

Nearly all of these foundation classes are various subclasses of the Container class for implementing various different types of data structures. Usually a dynamic data structure is defined much farther down the class hierarchy in an object-oriented system. This is a fairly innovative way of approaching the matter that has a lot to be said for it.

The Buffer class is intended to provide a simple means of storing and maintaining large blocks of data in either byte or word formats. An immediate extension to this class that suggests itself is to add an instance variable that indicates whether a given Buffer object is a byte or word buffer. General read and write routines could then be written that first send a message to a buffer to see if it is in byte or word format. This way, a program need not be specific to one type of buffer or the other.

The Stream class is implemented as a subclass of Buffer and includes one instance variable *position* as a way of indexing the current stream position. The Collection classes provide for objects in which other objects are grouped. The subclasses of Collection itself vary according to whether access to elements is ordered or random, and whether the size is bounded or unbounded. So, for example, the OrdCollect class allows elements to be inserted and retrieved in sequential order, and for the size of the collection itself to be expanded automatically as new elements are added.

Text Window Classes

Some additional classes have been developed by CNS that provide support

```
initialize
/* Initialize the reciever: set up MetaWindow, win_list and create root  */
/* window.   */
{ id      newWin;
  int     w, h;
  extern  id  Menu;

  @ScreenMgr create_ &Screen@;
  self->color = @Screen color@;
  w = @Screen getXmax@;   h = @Screen getYmax@;
  @Mouse create_ &Mcursor@;            /* init mouse service routine */
  @Mcursor setColor_ self->color@;
  @OrdCollect new_ &self->winList  size_ 25@;
  @Window new_ &newWin par_ NIL  x_ 0 y_  0 wdth_ w hght_ h@;
  self->rootWin = newWin;
  @self->rootWin getAbsRegion_ &mbox@; /* init movement box */
  /*@self->winList add_ self->rootWin@;*/
  self->activeWin = self->rootWin;
  Dispatcher = @Notifier create@;     /* an instance of the event manager */
}
```

Example 1: *Initializing the receiver*

for the creation of text windows in the C_talk environment. (See Table 2, this page.)

Let's look briefly at what some of these additional classes do. The Browser class provides some of the basic methods for creating a browser similar to the one included in the C_talk system. Over 20 methods are supplied with this class. Even a main program is included for creating a test browser program. This code can produce a "test browser" that lacks various things that the real tool has, but shows many of the essentials of how the tool is created. For those who want to develop their own custom browser that they can continue

```
Object
  Assoc
  Container
    Buffer
      Stream
    ByteArray
    Collection
      OrdCollect
        Stack
    Set
      Dictionary
    IntArray
    String
```

Table 1: *The foundation classes are listed in hierarchical format.*

to extend at will, this code will represent a flying head start in that direction.

The Browser class is a direct subclass of Object that makes use of the TxBuffer, TxEditor, Response, and File classes. The TxWindow class and its subclass TxEditor coordinate handling text in windows by sending messages to objects in nine different classes. Together they handle processing keystrokes and managing characters in buffers.

```
Browser
File
Menu
Mouse
Notifier
ScreenMgr
TxPoint
Window
  ButtonWin
    ItemWin
  Scrollbar
  StdWindow
    ListWindow
      HorListWin
    PopUp
  Response
  TxtWindow
    TxEditor
WinManager
```

Table 2: *The additional classes are arranged hierarchically.*

Window is the abstract class for forming most of the classes that are used for defining the various features of windows, such as those used by TxWindow. The WinManager class is the one that ties it all together. This class sends messages to most of the other classes so as to coordinate their behavior. Currently, it has a total of about 17 methods. A quick way to get an overview of what it does is to take a peek at its initialize method, which is reproduced here in Example 1, page 76.

The initialization routine first sends a message to the Screen Manager and tells it the type of screen to create. Then the Mouse object is told to create the specified type of cursor. Next an Ordered Collection object is created to serve as a window list. Then the Window object itself is created with the designated proportions and attributes, and finally an Event Manager is created. This is an excellent example of how C_talk makes use of one of the main paradigms in object-oriented system design, that of employing one object as the manager of other objects. The only shortcoming of this approach is that it tends to bog down when the number of objects involved gets large.

Runtime Applications

Completing a C_talk application is generally a five-step process. The first step is to write the necessary source code files in the C and C_talk syntax, which is written in disk files with the .PRE file extension. The second step is to write a main routine in C that calls all the necessary modules. Once this is done, the third step can be taken, which is to run the preprocessor and compile all the C_talk files into the corresponding C files. The fourth step is then to compile all the C files with a C compiler. Finally, the compiled .OBJ files are linked together to form the executable file for the application.

For those who wish, as mentioned earlier, there is a convenient way of collapsing some of these steps using the make option in the C_talk Browser. To do so, you would select the Make Spec pop-up menu. The result of using this interactive procedure is the creation of a .MAK file that can be used by a special auxiliary program that will automatically carry out the three steps of preprocessing, compilation, and linking as a batch process. And since the browser supports calling DOS commands, and exiting to DOS, it is not necessary to exit the browser to make the finished executable and even test it afterward.

Conclusions

C_talk succeeds very well in doing what it sets out to do, namely provide a language that is a true hybrid between C and Smalltalk. The browser puts it in a class all by itself among other similar attempts. One thing that should be pointed out is that a major difference between Smalltalk itself and all of the C hybrids that have appeared so far is the great difference in the number of foundation classes present. As it is sold, Smalltalk is not just a bare-bone language, but amounts, in effect, to a language plus a large generic function library that is standard in this case by definition.

For this reason, the Window classes we have described are an important addition for the C_talk system. The importance of these library classes can be appreciated when you recall that objects in an object-oriented system are working parts for programs rather than just stand-alone functions. In a sense, then, a system with a large class library is one with a substantial part of the programming already done. The downside is that the programmer has to learn these classes and how to use them. Although one of the advantages of the object-oriented approach is that this is all done generically, the learning curve still exists for class libraries.

One minor potential drawback with this type of system is that it is set up to work mainly with C_talk source code in the browser. If a developer wants to sell class libraries in the object file format, it will not be possible to use them with the browser in the current design. Perhaps a way can be devised for loading key class information into the browser without divulging all the source code involved. This would allow the browser to be used even with object code class libraries. The main thing to emphasize is that C_talk has all of the essential things of a true object-oriented language, and the browser has done well enough to give the interactive feel of a Smalltalk environment while offering all the advantages of any C-based system.

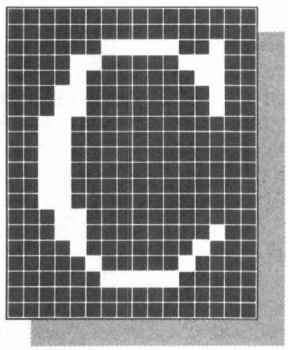
Product Information

C_talk, Version 1.4; CNS Inc., 7090 Shady Oak Rd., Eden Prairie, MN 55344; 612-944-0170. PCs and compatibles. Requires: DOS 2.1 or later; graphics card; and one of the following: Microsoft C, Lattice C, Turbo C, or C86. A hard disk and a mouse are recommended. Price: $149.95.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Context-Sensitive Help and TWRP, the Tiny Word Processor



by Al Stevens

Over the last several months, we have been assembling a library of window-oriented tools to be used in the development of a communications program. Last month's column featured a text editor engine that allows you to collect user-entered, free-form text into a buffer through a window. This month we will surround that engine with a menu structure and add file-management and text-searching functions. First, though, we must complete the collection of window tools by adding the context-sensitive help feature.

Most PC programs offer some type of on-line help to the user. This help usually takes the form of text windows that pop up and explain something the user needs to know. When the text automatically reflects the user's current position in the program, the help text is said to be "context-sensitive."

The context-sensitive help feature for our C Programming project is implemented in two source files. These are Listing One, help.h, on page 112, and Listing Two, help.c, on page 112. The code from previous months has included constructs to support the help functions. The data entry and menu functions in October provide for the identification of help windows to explain each menu selection and data entry field. The structures found in entry.h and menu.h already have places for the window mnemonics, and the *getkey* function (window.c, September *DDJ*) already watches for a help hot key and calls a help function.

Help windows are described in an ASCII text file. Each window has a mnemonic and text. The mnemonic is from 1 to 8 characters, and the text can be up to 78 characters wide and 23 lines deep. Each mnemonic is surrounded by angle brackets, appears at the beginning of a line, and is the only entry on the line. The text for a window follows its mnemonic and is terminated by the appearance of the next window mnenomic or the *<end>* file-terminating mnemonic. Listing Three, page 121, is twrp.hlp, a help text file that we will use in this month's expansion of the editor. Look at this listing now for an understanding of the idea. Later you might want to change the file to reflect changes you make to the editor commands when you customize the editor with your own preferred command set.

The *getkey* function watches two global variables —the function pointer named *helpfunc* and the integer named *helpkey*. If a program wants to intercept a help keystroke, it can assign a function address and a keystroke value to these variables. Then, if *getkey* senses that the user has pressed the assigned help key, it calls the assigned function.

The source file named help.c includes the function *load_help*, which an application program calls to load a help text file. The function assigns a help function key (F1 in our implementation) by initializing the *helpkey* integer and assigns its own help function (*display_help*) by initializing the *helpfunc* pointer. The file *help.h* contains a macro, *set_help*, that allows an application function to name the mnemonic for a help window. With this macro, an application can maintain the proper current window for the help context of the program. The source files entry.h and menu.h from October define structures that an application program initializes to use the data entry and menu functions. The structures include pointers to character pointers, which, if initialized, provide the window mnemonics for help windows for menu selections and data entry fields. The *display_help* function senses that an application has used *set_help* to establish a help window, that the menu cursor is on a menu selection, or that a data entry field is being processed. The *display_help* function thus selects the appropriate window if the user presses the help key. With this help facility applied along with the other window functions, the applications programmer can employ context-sensitive help windows for menus, data entry screens, and other program modes. The system developer needs only to do the following things to implement the feature:

1. Build a help text file.
2. Insert window mnemonics into the menu and data entry structures.
3. Insert *set_help* macros into the application program for help windows that are not related to menus or data entry fields.

The Tiny Word Processor

I promised you an expanded text editor, one that will support the communications program. For this editor, we will use the window editor function from November, the data entry and menu functions from October, and the context-sensitive help window functions just described. We can call this expanded editor from applications programs just as we can the less well-endowed editor engine of last month. To provide an example of the expanded editor, I built a simple front end that uses the editor in the way that our utility program will use it —as a tiny word processor. With the front end, the program becomes a stand-alone editor package with minimum word-processing features and capabilities.

This new program marks a milestone —I am using it to write this column. This drastic measure is a test to ferret out the hidden bugs. It is a risky endeavor because I stand to lose my work if this unproven editor crashes or fails to save my deathless prose correctly. For this reason I will be making lots of departures from typing to save the text and take a look at what was saved —the sanctity of my sanity included. This new program is the first time we use the tools in an actual application. As my own beta tester, I find things that need to be changed. The first modification I make is to the editor colors, making them more pleasing than the bland monochrome combinations in the window.h defaults. These selections are personal; you'll want to make your own.

Next comes a pleasant discovery. Much of my work is done on airplanes, and I use a bare-bones Toshiba T1000 laptop MS-DOS computer —low cost,

low weight, minimum capabilities. As it happens, these words are being written between Dallas and Phoenix. One of the drawbacks to this bottom-of-the-line machine is that it takes forever to load any of the full-functioned word processors from the single 3.5-inch diskette. But our new tiny word processor is a mere 30K and loads itself in a few seconds, thus solving a months-old problem. Now I have to deal with another problem. Whenever I haul out the T1000, set it up on the tray table, and get deeply engrossed, my neighbor in the next seat gets interested in the cute little machine. This brings the inevitable tap on the arm and the subsequent unsolicited computer discussion in which I hear about the traveler's office computers and how he knows nothing about them (but his son is a whiz). People always ask where I got the T1000, what it cost, and what I use it for. My pal Bill Chaney would grumble something about keeping breeding records for pit bulls, none of which would be true, but he wouldn't be bothered anymore and could then work undisturbed.

To build the tiny word processor, which I will now call TWRP (pronounced "twerp"), we surround the window text editor from last month with a menu shell and some file-management features. These features employ the menu and data entry functions from October. We want the editor to be available in two ways: as a stand-alone program and as a function that can be called from an application program. TWRP is built from two source files. Listing Four, page 122, is editshel.c. This file contains the menu shell and the advanced editor commands for file management and text searching. With editshel.c, an application program can link to the advanced editor features. Listing Five, page 124, is twrp.c, the outer layer that uses the advanced editor to build TWRP as a stand-alone program.

Notice that twrp.c contains the call to *load_help* to load the help file for the editor. By isolating this call in the outer function, we reserve the ability for other programs that call the editor to have other, perhaps more complex, help files.

editshel.c contains a set of expanded commands as *case* statements in a *switch* statement. The function that contains this *switch* is named *editmenu*. This function represents the technique we will use to add features to the editor as they are required. When editor.c encounters a key stroke that it does not recognize, it calls the function addressed in an external function pointer named *edit_func*. If the pointer is *NULL*, the editor rings the bell to indicate that the command is invalid. By initializing that pointer, we can extend the command set. By putting a similar function pointer chain at the end of our extended command set, we can keep the door open for future extensions. It would be easier to add *cases* to the original editor's command-dispatching *switch*, but then we would need to publish a modified editor.c each time we added features to the editor. With this approach each extension is contained in the source module that has the extending code. Another advantage of our extension strategy is that it gives the editor multiple levels of functionality. The current level depends on the context in which the editor is executed. A program that uses the expanded editor can also use the simpler window editor to collect small amounts of text for other reasons. That invocation of the editor would not, for example, have file-management commands because the extension function pointer would be *NULL* for the duration of the editing session.

Because we are discussing extended editor commands, we should consider what the big-time word processors do that TWRP cannot do. TWRP has no spelling checker or thesaurus, no keyboard macros, no fonts, no columns, no index or table of contents generator, no style sheets, no footnotes. It sounds a lot like the WordStar of yore. But wait, there's more —or less, depending on how you look at it. TWRP has no underscoring, boldface, or subscripting and superscripting; no pagination; no mail-merge; no headers and footers; no printing; no variable margins.

So what good is it, and what does it do? TWRP is primarily a text-composition tool, supporting the most basic text entry activities needed by writers of everything from memos to manuscripts. You type the text in, and TWRP saves it for you. You can load and save text files; merge text from other files; form paragraphs; mark blocks; and move, copy, and delete blocks.

TWRP is a lot like the popular desktop-accessory notepad programs. It has, however, one unique and endearing quality —it can be linked into our programs as an in-line word processor, fully integrated with whatever else we have going on. What's more, the source code is ours and we can make it do whatever we like.

This observation returns us to the subject of macros. TWRP has no keyboard macros in the style of many editors and word processors. Brief, the world class programmer's editor from Solution Systems, has a macro language that resembles Lisp. The ME editor from Magma Software Systems and its derivative, the New York Word Processor, have macro languages that resemble C. The new Microsoft Editor uses compiled C extensions for advanced macros. Borland's Sprint word processor has a procedural macro language. So does XyWrite from XyQuest. All these editors and word processors have compiled or interpreted macro languages, some of which resemble C.

In that spirit, therefore, TWRP is, by declaration and acclamation, blessed with a macro language that not only resembles C, it is C. If you want to add a macro, you add an extension function in the manner described previously, write the procedure as a C function, and recompile and link TWRP. What better way for a C programmer to add a complex macro to an editor?

The MAKE Files

Listing Six, page 124, is twrp.prj, the Turbo C project make file for the Turbo C environment compiler. Use the compact-memory model with all error checking enabled. You will need source files from the September, October, November, and this month's C Programming column.

C Tool Set Errata

My use of the tools has uncovered a few minor problems that I will address here. When I moved to the compact-data model to support an 800-line edit buffer, a bug showed up in window.c (September *DDJ*). The *writeline* function has some nonportable pointer juggling that needs to be tightened up. Substitute these lines for the call to the *__vram* function:

```
__vram(__vptr(x+wkw.lf-1,
                        y+wkw.tp-1),
    (int far *) cl,
    (unsigned) ((int far *) cp -
                        (int far *) cl));
```

The *menu select* function in menu.c failed to reset the global pointer named *mn* to *NULL* before returning. Insert this statement before the return from *menu select* at the bottom of the function:

```
mn = NULL;
```

I will update the download versions of these files on CompuServe to reflect these corrections.

Turbo C 2.0

All the window tools from the C Programming columns have been built with

Turbo C, Version 2.0. This is a last-minute development; 2.0 has not been out long. The big news is the integrated debugger in the Turbo C environment; everyone has been waiting for that well-rumored feature for a long time, and Turbo C has now removed the last reason for feeling inferior to QuickC. I have spent a lot of time using the debugger and can say without reservation that you will be delighted with it. I am especially impressed by its seamless integration with the editor and compiler environment. Our small TWRP project is by no means an exhaustive test of the compiler, so this endorsement is preliminary but enthusiastic.

Turbo C has a new pop-up utility program named THELP that uses the Turbo on-line help file. This is a significant addition to the package. You know how Ctrl-F1 works in the TC environment—it pops up a context-sensitive help window about Turbo C. Now you can have those same language and compiler help displays from within your personal text editor (including TWRP). The THELP hot key pops up the Turbo help windows over whatever you happen to be doing. If the cursor is on a Turbo keyword—a library function name, perhaps—the help window related to that particular word is displayed.

TesSeRact

THELP was built from the TesSeRact shareware memory-resident program library. This library is available in files you can download from CompuServe in the CL and Borland libraries and from other BBSs. Source code is available to registered users.

TesSeRact is said to be the answer to a terminate-and-stay-resident (TSR) programmer's prayer, able to make any program a well-behaved TSR capable of running in harmonious accord with other programs, TSR and otherwise. I don't completely believe this, but if you want to write TSRs without understanding how they work, this is a good way to go. The documentation explains how to use the libraries, which are available for C, assembly language, and Pascal. To use the libraries in programs that you distribute, you must register your use and keep the TesSeRact signature in the code. That's how I figured out that THELP is a TesSeRact program.

The developers of TesSeRact are experts in the black art of TSR programming. The development team is what remains of the Ringmaster project, a failed industry attempt to settle on a standard for TSRs. Given a lack of enthusiasm among private software publishers to sign up to such a standard, some of the team splintered off and took the shareware approach. The libraries that resulted provide a reasonably safe way to get a TSR program running with a minimum of fuss. Having researched, developed, and written extensively about TSR programs, I am convinced that there is no way to write one that can be guaranteed to work in all PC hardware environments and in the company of all other programs and systems. MS-DOS just wasn't meant to be used that way, and all those successful TSRs are the legacy of a generation of hackers who figured out ways around the limitations of DOS—even if a lot of the programs don't work with one another. TSRs, working or not, will be with us as long as MS-DOS is with us, and that is going to be a long time.

A Crotchet: The Trouble with Programming

The trouble with programming is that it isn't getting any easier. Quite the opposite, programming is getting a lot harder.

In recent months I have been looking at new or different software development environments. In a recent column I told of my first exposure to C++ and object-oriented programming, a most humbling experience. Another venture into the unknown (to me, that is) was my excursion into the Macintosh. Someone observed that it takes a good programmer at least a year to become a competent Mac programmer in C or any other language.

Now I am unfolding the OS/2 Software Development Kit (SDK) from Microsoft. Seventy-five pounds of manuals and diskettes and eight videotapes comprise this formidable beast. At first glance it appears that unraveling what manuals contain what information is itself a monumental effort, not to mention the learning process that follows. I have only just begun, and I am wondering how many blue moons will pass before I see an OS/2 screen group displaying my very own "Hello, world" message.

No, programming isn't getting any easier. Remember the promise of application generators and fourth-generation languages that were going to put system design and programming within the reach of Everyman? A new wave of dBase and NOMAD pseudoprogrammers were going to put us all out of business. Not to worry. The complexity of these new software development platforms ensures a place for hacker-level talent for years to come.

The best part of this gradual evolution to more complicated programming is that we don't need to learn a new language. C is surviving and is still that overworked cliche, the "language of choice"—not that it will be easy to remember the 900+ OS/2 function calls, but at least their basis is in the syntax of the best programming language ever devised.

It does, however, raise my C purist's hackles when I see the function declarations assigned the *pascal* type in their prototypes. It is fortunate that most of those blights are hidden away in header files. Indeed!

A Book for All Seasons

When you tackle OS/2, plan to spend some money at the bookstore. You will have to augment the 75 pounds of abstruse Microsoft manuals with a few more readable offerings from other authors. I have most of the books that have been written about OS/2, and one stands out as the essential first book to read. Its title is *Inside OS/2*, and its author is Gordon Letwin. Do not confuse this book with another one with the same title. This one is published by Microsoft Press, and to its credit, Microsoft includes a copy of it with the SDK. *Inside OS/2* is not the only book you will need, but it is the first. Read it even before you watch the videotapes. That way, if you sleep through something important, chances are Letwin has already explained it.

Inside OS/2 is an excellent description of the underlying design principles for OS/2. It will help if you already know a little bit about operating system theory. An understanding of the concepts explained in this work is essential for the design of programs that would effectively use the advanced features of OS/2. I like this book.

(The little old lady in the next seat is eyeing my T1000. Soon she'll be tapping me on the arm. Where are you, Bill Chaney, when I need you?)

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 112.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Cross-Referencing Pascal and Modula-2 Programs

by Kent Porter

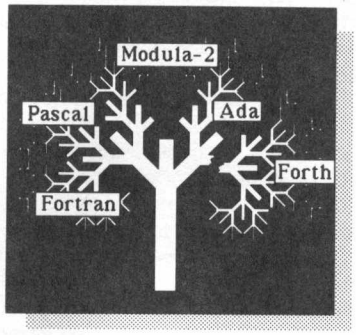
The SUB.PAS utility that I did for the August issue of *DDJ* drew favorable comments from the esteemed readership. Compliments are always good for my ego (keep 'em coming!), but, more important, they let me know what you want. Because you obviously like utilities, here we go with another one, this one specifically targeted toward programmers.

It's a symbolic cross-reference utility for Pascal and Modula-2 programs. XREF prints an alphabetized listing showing the line numbers where all identifiers occur, an "identifier" being either a symbolic name or a language keyword. Because of the syntactic similarities between Pascal and Modula-2, the same program fits both languages, with only minor nods in one direction or the other. You can use the cross-reference listing to find all occurrences of any identifier, and also to spot variables declared but never referenced. That's its utilitarian side.

From a more theoretical viewpoint, XREF implements some unusual dynamic data structures that have applications in compilers and other text-and token-processing programs. There's probably a definitive name for these data structures buried in Knuth or some such. I call them "Binary B+ Trees" for reasons that will become apparent later.

Because of space limitations the version of XREF published in Listing One, page 126, is written in Turbo Pascal. There's also a version written for JPI's TopSpeed Modula-2, called XREF.MOD. It doesn't appear here, but you can get it by downloading from CompuServe or ordering a diskette from *DDJ*. There are only two significant difference between the Pascal and Modula programs other than the source languages. One is that the Pascal version is not case sensitive by default and the Modula-2 version is. This is consistent with the languages themselves. The other is that the Pascal version assumes a filename extension of .PAS if none is explicitly furnished, but the Modula program makes no assumptions. Again, this is consistent; Pascal source files typically have the .PAS extension, whereas Modula source files might have .DEF or .MOD extensions.

The full command line for the Pascal version is thus



XREF <Filepath.[Ext]> [/C|/N]

and for the Modula-2 program it's

XREF <Filepath.Ext> [/C|/N]

The optional switches /C or /N specify case sensitivity and non-case sensitivity, respectively. By coercing the utility into non-case sensitivity, you group identifiers such as myVar and MyVar (equivalent in Pascal but not in Modula), receiving a cross-reference listing for MYVAR on the final output. The /N switch thus results in a slightly shorter cross-reference listing for long programs, especially those written in Pascal.

Listing One shows a corollary utility, NUMBER.PAS. The format of the listing shows exactly what NUMBER.PAS does: inserts a line number before each source line. This program furnishes a listing that makes it easy to use the cross-reference shown in Listing Two, page 126, the output for NUMBER.PAS produced by XREF using the command line

XREF NUMBER.PAS /C

For example, you can see that the identifier Len (initial cap) occurs in line 23 of NUMBER.PAS, and len (all lowercase) appears in lines 11, 24, and 27. A similar situation exists for identifiers Nbr and nbr. Had I specified /N on the command line —the default in the Pascal version —the cross-reference listing would have grouped occurrences of Len and len, and also of Nbr and nbr. The report heading would also have been different.

Note that the utility includes some language keywords but not others. For example, it shows occurrences of WORD and NOT, but excludes FOR, WHILE, BEGIN, and END. The rationale is to apply a selective commonsense filter that gets rid of nuisance keywords while showing those you might be interested in finding. Note, as an example, that lines 268-282 in Listing Three, page 127 —the XREF.PAS source program —eliminate the keyword TO used in FOR loops, but not DOWNTO. The reason is that decrementings are unusual, and you might want to find all loops that count down by cross-referencing the DOWNTO keyword. If you disagree with my list of nuisance keywords, you can easily modify the XREF source accordingly.

Now let's look at the internals of XREF, which are the parts that make it not only useful but interesting from a programmer's viewpoint.

How It Works

The main body for XREF begins at line 400 in Listing Three. The flow is broken down into several steps: initializing global variables, processing the command line, opening the file, announcing the program, and processing the file. There's nothing surprising in any of these macro steps.

The wrinkle occurs at line 454, where XREF decides what to do according to whether or not the cross-reference is case sensitive. Let's consider the implications. If case is not an issue, as in Pascal, all identifiers are reduced to uppercase and are thus implicitly in alphabetic order. With case sensitivity, however, the ordering of identifiers is complicated by the ASCII collating order. Identifiers that start with an uppercase letter always precede those that begin with a lowercase letter. Thus, *Zulu* comes before *alpha*. Makes sense to the computer, but the human mind rebels.

For that reason the program branches at line 454. If case sensitive, it re-alphabetizes the symbol table using its symbols shifted to uppercase. This resorts the table according to absolute alphabetic order, not ASCII collating sequence. XREF then reports the results in an order such as able, Across, . . . , whisky, WORD, zebra, Zulu. If not case sensitive, XREF simply lists the identifiers in the existing order (for example, ABLE, ACROSS, . . . , WHISKY, WORD, ZEBRA, ZULU).

Let's delve further inside and see what's going on.

Of Parsing and Tokens and Such

Writing a symbol cross-reference utility is somewhat like writing a compiler. You have to pull the lexical elements —keywords and identifiers —out of source text lines, and then organize them in some meaningful way that makes it easy to refer back to them later. A compiler has to pay more atten-

(continued from page 86)

tion to detail than XREF does, of course, but there's still much in common between the two. And the same principles apply to writing HyperText data bases and other text-manipulating programs.

Plucking text elements from source lines is called parsing, and the resulting elements are often referred to as tokens. For example, let's say we have the source line

```
IF This < That THEN
```

For our purposes, this line contains four text tokens that the parser must produce: IF, This, That, and THEN. The fifth token is the less-than operator; a compiler would pay attention to it as well, but for a symbolic cross-reference it's ignored.

A parser operates according to rules that define what is and is not a data element of a specific category. In effect, it feels its way through a line of text, deciding what to do next on a character-by-character basis. For example, if the parser encounters a space at the start of a line, it feels its way through successive characters until it finds a nonspace. At that point, it switches modes, accumulating characters of a specific class until it finds a character outside that class: say another space. When that happens, it returns the token it has assembled. The parser's caller then has to decide what to do with the token.

In Pascal and Modula-2, characters of a specific class can be represented by sets. For example, the characters that constitute symbols are made up of the digits, letters, period, underscore, and caret (pointer indicator). Line 34 in Listing Three specifies them as the SymChars set. The Token function tests characters for inclusion in this set at lines 132, 169, and 178. If the character being "felt" is not in the SymChars set, then it must not be part of a valid token. A failure of the test triggers some appropriate action, such as sending the token accumulated to date back to the caller. That's what happens in line 178; it sends the token back to the caller at line 267, which assigns it to the variable Symbol and initiates other processing.

Source programs almost always contain comments and quoted literals. Because these are not symbols, it's necessary to skip over them, lest the cross-reference listing become cluttered with irrelevant information. The token parser does this by watching for the bookend symbols that open and close comments and quoted literals. For example, a Pascal comment can begin with a left curly brace. When the parser finds one, it scans the intervening text until it encounters a right curly brace, indicating the end of the comment. It behaves similarly for a literal enclosed by single or double quotes.

The (* and *) comment brackets used in both Pascal and Modula-2 are a little more complicated, but not much. On finding a left parenthesis, the parser checks to see if the next character is an asterisk; if so, it begins searching for another asterisk followed by a right parenthesis. The scanning operation is performed by the FindEndOfComment procedure, which receives a parameter specifying the end-of-comment character to look for.

Because comments can be nested, the Token procedure tracks the nesting level using the CommentLevel variable. Ordinarily this variable has a value of 0 to indicate that the parser is outside any comments or quoted literals. The value goes up and down as the nesting level changes; as long as it's nonzero, all text elements are ignored.

Valid symbols and the line numbers where they occur go into an elaborate dynamic data structure.

The Binary B+ Tree

Binary trees are two-way branching data structures that lend themselves to fast searches and the inherent ordering of data elements by the key value (Figure 1, page 89). At any given node, all the nodes conceptually to the left have keys of a lower value, and all nodes to the right have higher-valued keys. B Trees (Figure 2, page 89) are different. Here, each node consists of a variable-length list of pointers, each pointing to a lower-level object that is either an end node (a "leaf") or another list. A familiar example of a B Tree is the DOS directory structure. A B+ Tree is a minor variant, the difference being that the list elements also point to one another, forming a linked list. A Binary B+ Tree combines both.

Here's how it works in XREF. For each new symbol found by the parser, an object of type SymTreeNode (lines 23 – 29 in Listing Three) is inserted into the binary tree indicated by the global Head pointer. The BNode function starting at line 186 finds the insertion point by comparing the new symbol with those already in the tree, and branching left or right accordingly. The comparison is based on ASCII collating sequence, which separates lowercase identifiers from equivalent uppercase identifiers if the cross-reference is case sen-

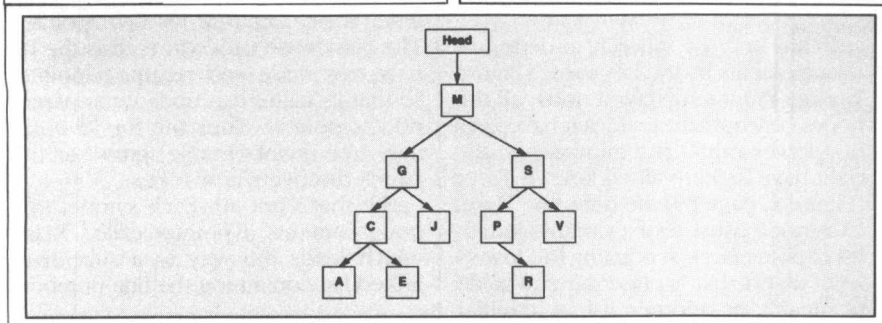


Figure 1: *A binary tree inherently organizes data in left-to-right order by key value*

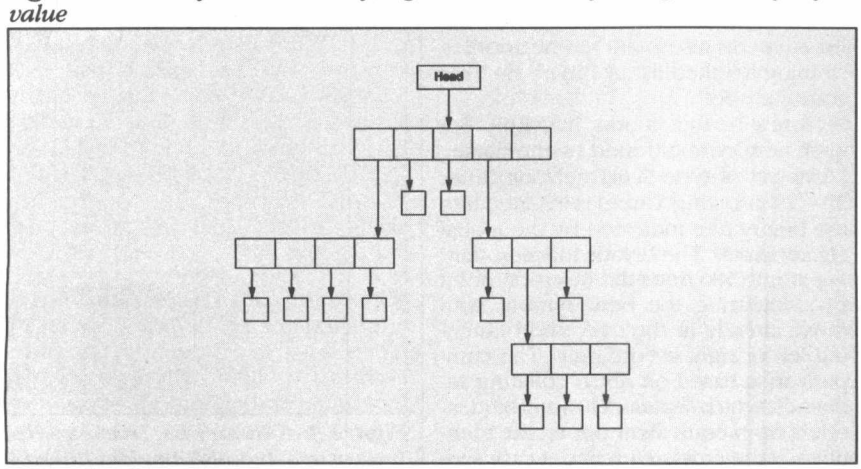


Figure 2: *A B Tree consists of variable-length lists of pointers indicating either other lists or end nodes*

sitive. If the new symbol is not already in the tree, the point of insertion occurs when the search reaches a NIL pointer. The NewNode procedure adds the binary tree node and returns a pointer so that its caller can update the parent node's pointer. Thus the top-level binary tree (symbol table) grows as the parser discovers new tokens.

But that's not all. Each symbol tree node contains a pointer called XList, which leads the way to a subsidiary linked list containing the line numbers where the symbol occurs in the source listing. When a new symbol node is being created, the Append procedure starting at line 213 begins the line number list by allocating a node of type XLineNode (line 17), which contains the first line number where the symbol is found. Append notes the address in the owning symbol tree node.

Now let's say that the parser finds the second occurrence of a particular symbol. In this case, BNode finds a match as it searches the tree and returns a pointer to the node for that symbol. Append gets control again, but this time —finding that the line-number list already exists —it searches the singly linked list until it finds the tail, indicated by a NIL pointer in an object of type XLineNode. It allocates a new node of this type, stores the line number, and updates the old tail's Next pointer.

The Binary B+ Tree thus assumes the form shown in Figure 3, page 90, where each symbol node potentially points to its left and right neighbors and also to its line-number list. This structure makes for fast searches and updates, assuming that the tree is fairly well balanced. In most cases it will be, because identifiers occur in alphabetically random order in source text. The structure also offers the advantage of automatically sorting symbols into ASCII order.

Traversals of a Binary B+ Tree are the same as the recursive techniques employed in connection with binary trees. For example, to do something with nodes in lowest to highest sequence by key value, you can use in-order traversal as follows:

```
PROCEDURE InOrder (Node :
                          NodePointer);
BEGIN
   IF Node <> NIL THEN BEGIN
      InOrder (Node^.Left);
      (* Do something with this node*)
      InOrder (Node^.Right);
   END;
END;
```

The first recursive call forces the program to strive left, visiting successive nodes until it reaches a NIL pointer. The stack keeps track of the order in which nodes are visited. Upon hitting a NIL pointer, control backs up to the most recent node and does to it whatever is required. The second recursive call then moves to the right from this node, where the leftward striving begins again. Thus all the nodes are visited in lowest to highest order by key value. Traversal ends when every NIL pointer has been found.

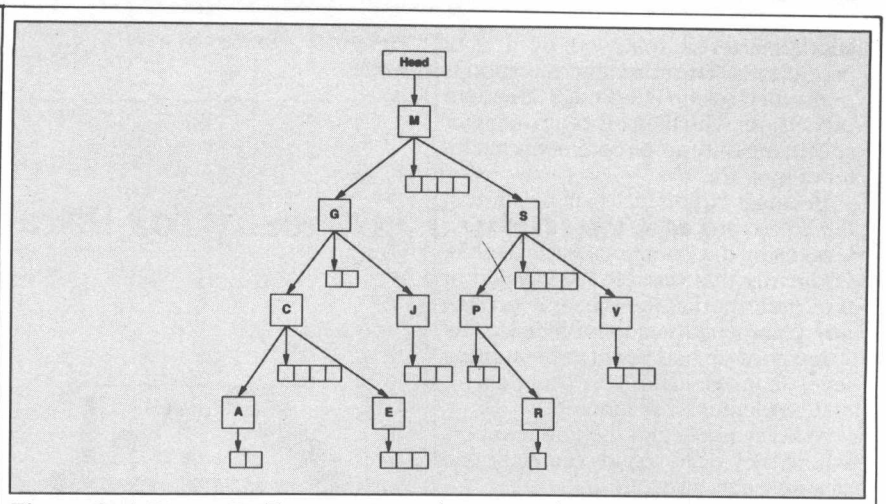


Figure 3: *A Binary B+ Tree organizes data according to the key value like a binary tree, but also employs linked lists in the manner of B+ Trees*

The Report, Alphabetize, and Count procedures in Listing Three are all based on this in-order traversal model. The body of Alphabetize (lines 379 – 386) adds another wrinkle by disposing of the symbol tree node just before leaving it for the last time. Doing something to a node after working on all its left *and* right children is called postorder traversal.

And what purpose does Alphabetize serve? You'll recall that the symbol tree is ordered by ASCII sequence, which means that *Z* comes before *a*. In non-case sensitive mode all symbols are shifted to uppercase, so it doesn't matter. If case sensitivity is required, though, we need to resort the table so that *able* comes before *Acme* as the mind expects, and not somewhere toward the end of the printout. That's what Alphabetize does. It structures a new symbol tree based on the alphabetic order of the uppercase equivalents of each symbol. This new tree replaces the old one, which is why the body of the procedure disposes of the old nodes after copying their contents. The work is now done except for printing the results.

Getting the Cross-Reference Listing

The Report procedure performs an in-order traversal of a Binary B+ Tree: either the original indicated by the global Head pointer, or the reordered one pointed to by Alpha. The IF logic starting at line 454 decides how to handle the situation depending on the *Case_Sensitive* switch.

Report is chiefly a printer control routine that recursively visits the tree nodes in order, thus producing an alphabetic listing. It prints the symbol and, in parentheses, the number of occurrences. It then follows the singly linked list associated with the node, printing each line number. The *Col* variable keeps track of the horizontal position and triggers a new line when the current line is full. The global LineNumber variable, no longer needed to track the source line, is pressed back into service to control pagination of the printed output. The Count procedure runs through the symbol tree one last time, yielding a symbol count that appears at the end of the report.

One final point: If the comment level is not zero at the conclusion of the parsing and symbol table creation, you have either a runaway comment or an unterminated string somewhere in the source. The program reports this problem at line 450; it's equivalent to an "Unexpected end of file" error message from a compiler.

So there it is, a handy utility for Pascal and Modula-2 programmers, as well as an intriguing dynamic data structure that you can apply to all sorts of problems that require the ordering of complex information.

Availability

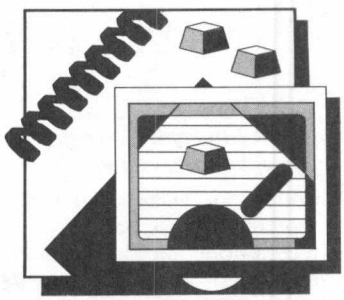
All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings start on page 126.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

Developing Taxonomy for Parallel Algorithms



For as long as I can remember, I've been fascinated by the issue of understanding.

As long as I don't ask myself to define my terms too finely, I can convince myself that I sort of understand how we understand. Surely, I tell myself, understanding is a process of reducing surface complexity without losing the information in the depths. Identifying minimal-surface-area packing schemes for information. Something like that. And I even think I know some of the techniques we use when we attempt to understand something.

There's abstraction, there's seeing parallels, there's organization. In particular, we discover organization in —or impose organization on —the data: We rank and file the raw phenomena, we identify the properties and impose a taxonomy. This month's column is about an effort to taxonomize algorithms for parallel processing.

Imposing a Property Taxonomy

Two years ago computer scientists from several universities organized a workshop in Santa Fe, New Mexico, to discuss the possibilities of developing a taxonomy of parallel algorithms. What the organizers came away with from the workshop was not a taxonomy, but an increased respect for the size of the taxonomic task.

by Michael Swaine

A neat array of pigeonholes in which to put parallel algorithms, the workshoppers concluded, was an edifice that the field of computer science was not yet ready to erect. First some spadework would have to be done. There were at least these preliminary problems: 1. Identifying the properties or attributes of algorithms that actually had something to do with parallelism. 2. Untangling the relationships among problem, algorithm, and architecture in parallel processing. 3. Clearing away some of the inconsistencies in terminology that arose from the fact that parallel processing was really several different paradigms.

Some workshops produce proceedings. (And some publish the proceedings before the workshop has taken place; these documents should perhaps be called precedings. But I digress.) This workshop brought forth, after a gestational hiatus, an interesting book, *The Characteristics of Parallel Algorithms*, Leah H. Jamieson, Dennis B. Gannon, and Robert J. Douglass, editors; The MIT Press, Cambridge, Mass., 1987. The book is not the patchwork proceeding of a workshop, but a well-organized building plan for a taxonomy of parallel algorithms.

The book consists of three sections: The first attempts to identify characteristics of algorithms relevant to parallel processing, the second examines the characteristics of concurrency from the point of view of applications, and the third looks at how a taxonomy of parallel algorithms would affect the organization of software engineering tools for the programmer. I'll discuss here some of what the authors attempt in the first section.

The book opens at a paradigmatic level. In the past this column has generously given out several definitions for the word *paradigm*, and will not hold back on offering another: For present purposes let's let the word *paradigm* —or better, the expression *the paradigmatic challenge* —refer to the mapping of problems onto algorithms and architectures. That's a little ethereal, but the book is not all theory. It digs into the analysis of parallel algorithms, the characterization of parallel algorithms and architectures, and the benchmarking of parallel architectures.

Three Paradigms

Philip Nelson and Lawrence Snyder of the University of Washington open the book with a discussion of the importance of the paradigmatic perspective for getting some understanding of parallel processing. They say "In the case of parallel computation . . . paradigms generally encapsulate information about useful communication patterns." In other words, the paradigm selected specifies the useful patterns of communication, and efficient communication is precisely the problem in parallel processing architecture.

Nelson and Snyder consider three broad programming paradigms: divide-and-conquer,compute-aggregate-broadcast (CAB), and the systolic and pilpelined paradigm.

Divide-and-conquer was probably first cogently described as a programming paradigm by Robert Floyd in his Turing Award lecture, "The Paradigms of Programming." (You can read Floyd's description in *ACM Turing Award Lectures: The First Twenty Years: 1966-1985*, ACM Press/Addison-Wesley, 1987.) The idea is that the problem is divided up into two or more smaller problems to be solved independently, and the solutions are combined. If the smaller problems are just smaller versions of the original problem, the paradigm is recursion. Nelson and Snyder shed new light on divide-and-conquer as a parallel processing paradigm and attempt to identify just how it differs from other parallel processing paradigms.

One conclusion they draw is that the

divide-and-conquer paradigm is often (although not always) characterized by a binary n-cube communication structure. (Of course, given a predisposition to see communication structure as the problem in parallel processing, it is not surprising that it is communication structures they pull out as the distinguishing marks of individual parallel processing approaches. I'm not suggesting that Nelson and Snyder are wrong in this, but just pointing out one example of how your paradigm conditions the questions you think to ask.)

Algorithms in the CAB paradigm have three phases: a compute phase; an aggregate phase, in which local values are combined from processes into (fewer) global values; and a broadcast phase, which returns global values to the processes. The authors look at several such algorithms, but I was unable to find a general conclusion that they drew about the paradigm. Maybe I just missed it.

Nelson and Snyder lump systolic and pipelined approaches into one paradigm, characterized by "the decomposition of the problem into subcomputations that are assigned to dedicated *processes* with the data 'flowing' through the processes, visiting all or an appropriate subset of processes to complete the computation for that input." One example algorithm is the band matrix multiplication algorithm of Kung and Lieserson. The "systolic and pipelined paradigm" may sound more like an architecture than a paradigm. Indeed, parallel paradigms are tied tightly to architectures as well as to algorithms, and sometimes you wonder which is the cart and which is the horse. As we'll see momentarily, some of the book's contributors directly attack the problem of the relationships among paradigms, algorithms, and architectures in parallel processing.

The distinguishing characteristic of the systolic and pipelined paradigm seems to be the idea of flow, a (there you go jumping ahead of me again) communication property. The authors present a rigorous test of the "flow property," which, unfortunately, does not always identify putatively systolic algorithms as having the flow property. Nevertheless, it appears to test an interesting property common to many such algorithms.

> The flow test works like this: You select an arbitrary communication edge and "radioactively tag" a single communication across that edge. As the computation progresses . . . let all values computed with one or more radioactive values become radioactive. Then define the "contaminated region" as the set of processes and communication channels touched by some radioactive value. An algorithm passes the flow test if the edge over which the initial value was transmitted is on the boundary of the region; otherwise it fails the test.

The idea is that the tagged value flows out to define the contaminated region.

Mapping Algorithms to Architectures

It seems that if we could neatly map algorithms to architectures, we could reduce the dimensionality of the paradigmatic challenge. Because we had defined the word *paradigm* to mean the mapping of problems to algorithms and architectures, the challenge then would be one of mapping the problem to the combined algorithm/architecture rather than one of solving a three-body problem. In fact, Leah Jamieson, a Purdue University computer scientist and the book's lead editor, talks about just this: how to map parallel algorithms to parallel architectures.

Jamieson introduces an interesting model, involving the algorithm life cycle and virtual algorithms. And she identifies the characteristics of parallel algorithms that she thinks are worthy of attention in taking a paradigmatic approach to parallel algorithms:

- type of parallelism (data or function)
- data granularity (size of data items processed as fundamental unit)
- module granularity (amount of processing that can be done independently)
- degree of parallelism (minimal to massive),
- uniformity of operations
- synchronization of requirements
- static/dynamic character of algorithm
- data dependencies
- fundamental operations required
- data types and precision
- the "natural" data structure for the algorithm, if any

But the performance of an algorithm depends on the architecture on which it is implemented.

Benchmarking Parallel Architectures

If, as Nelson and Snyder maintain, communication is the parallel processing problem, then the analysis of parallel algorithms requires some means of measuring communication performance in parallel implementations —both existing implementations and those implementations we'd build if we knew that the payoffs in performance would justify the costs. But the communication performance of a parallel algorithm will depend more (and in more complex ways) on the machine architecture on which it is realized than is the case for sequential algorithms on sequential machines.

Steven Levitan of the University of Massachusetts says that there exists no general theory of communication complexity, meaning that we can't predict performance in parallel processing. We can observe performance in real implementations, but we can't predict performance in architectures not yet built. We can't model; we must experiment. That's why the literature on parallel processing contains so many experimental case studies: "We implemented the following algorithm on the following architecture and obtained the following performance speedup over the baseline performance of the best-known sequential algorithm on a single-processor version of the same architecture. The experimental subject was then sacrificed and its cortex weighed. . . ." Excuse me; a momentary rat lab relapse.

Communication complexity combines components of algorithm complexity and architecture complexity. Levitan wants to separate the warp from the woof, to characterize the complexity component attributable to the architecture so that he can factor it out. To do this he needs to develop a metric for an architecture's communication complexity, and that's what he sets out to do in his chapter.

He begins by presenting several candidate metrics, including:

- diameter (worst-case time for a message to get from one node to another)
- bandwidth (maximum number of messages that can be sent simultaneously)
- path count (the number of paths through each node)
- narrowness (worst-case bottleneck when the network is partitioned in all possible ways)

He tests the metrics on some typical algorithms and gets mixed results. Some algorithm/architecture combinations are bandwidth limited, some diameter limited, and so on. Some perform in ways not well predicted by any of the metrics.

What he does next is interesting: He runs the algorithms themselves as metrics, taking care not to include the case in which an algorithm is used both as the measuring device and the object measured, and finds the algorithms uniformly better average predictors of machine performance than any of the abstract metrics. He concludes that the best metric for the problem is a good

suite of benchmarks, covering fundamental parallel processing tasks, and he presents one.

His suite of benchmarks consists of these tasks:

- broadcasting (necessary for any parallel algorithm to coordinate tasks)
- reporting (necessary for coordination and control)
- selecting (such as finding the maximum; the point is to collect data from all nodes)
- sorting (tests the ability to handle arbitrary communication patterns)
- propagating (same as testing the transitive closure of a graph)
- saturating (each node messages every other node; this tests bottlenecks or congestion)

The Analysis of Paradigms

Can we, Nelson and Snyder ask (or seem to me to be asking, because it's a question very much on my mind as I read their chapter), develop a computer science discipline that deals with the analysis of paradigms in a way that is analogous to the analysis of algorithms? They think so and suggest that such a discipline would teach techniques such as contraction analysis, which enables you to solve a problem for the paradigm and have the solution work for all algorithms that follow the paradigm. Here's Jamieson, talking about algorithms, and in the process arguing that paradigms can be identified by shared characteristics of algorithms, and therefore can be analyzed:

> The rationale behind the "characteristics-based" approaches is that many algorithms do possess an identifiable structure. For example, at the data dependency level, many algorithms share similar communications patterns. At the process level, algorithms based on the same paradigm —that is, divide-and-conquer —may exhibit similar communications requirements. Algorithms that operate on similar data structures may lend themselves to execution on similar architectures. [I]n the area of digital signal processing, it is possible to identify a set of canonical algorithm structures (that is, second order section, FFT, autocorrelation, convolution) and that algorithms that can be expressed in terms of these structures can be constructed from a set of basic building blocks (e.g., multiplications, complex multiplications, butterflies, sums-of-products, address arithmetic). A concise description of the algorithm in terms of a basic set of features allows selection of an appropriate machine configuration and can facilitate the mapping process by relating the characteristics of the current algorithm to known layout patterns.

The paradigms examined in the book are mostly familiar. Nelson and Snyder acknowledge that those they discuss do not provide the parallel programmer with a full toolkit of paradigms to guide him or her in developing new algorithms. But they conclude their chapter of the book by saying, "We expect more paradigms to be discovered." But that's an issue of a different context.

The Context of Discovery

Just fifty years ago the philosopher of science Hans Reichenbach drew a distinction between what he called the context of discovery and the context of justification in science. The former deals with the origins of theories; the latter takes theories as existing artifacts and investigates their properties.

The fact that Kepler arrived at his views on the structure of the solar system from a consideration of parallels with the Holy Trinity, Reichenbach maintained, was a fact in the domain of history or possibly of psychology, but definitely not in the domain of science or the philosophy of science. It belonged to the context of discovery, and science (actually he used the word *epistemology*) is concerned only with the context of justification.

As difficult as it may be to develop a rigorous computer science course on the analysis of paradigms, it's at least a challenge within the context of justification. Much less likely to be rigorously codified in the near future is the issue of how we discover a paradigm, map the paradigm to the problem, see the applicability, find the new rules. This ability resides in the context of discovery, the domain of serendipity and sudden insight. It's the art of the science.

It's easy to convince yourself that there are no rules for discovering rules, that seeing the connections is all luck or unanalyzable genius, that the art of the science is outside science, that this art is really artlessness. It's convenient to say that perhaps the Holy Trinity is as reasonable a place as any to look for inspiration in developing programming paradigms.

But this is a mistake. The context of discovery is accessible to the study of psychology, and although there may be some debate about the extent to which psychology is a science, it is at least a discipline with rules. As George Polya says, "There are rules and rules."

Polya has written the definitive book on discovery in problem solving (*Mathematical Discovery*, two volumes, John Wiley & Sons, 1965). In this extraordinary work Polya discusses very broad paradigms for problem solving and sets down the ten commandments for nurturing serendipity.

"Solving problems," Polya says, "is a practical art, like swimming, or skiing, or playing the piano: you can learn it only by imitation and practice." Polya presents in his book several examples for imitation, and discusses how to discover the pattern in example solutions, so that you can apply it to similar problems. He presents several such useful patterns, and by *pattern* he means nothing more or less than paradigm. He identifies several mathematical paradigms, but also such broad problem-solving paradigms as problem reduction (two kinds) and guess-and-test.

But Polya goes beyond merely identifying problem-solving paradigms; he also enters the context of discovery and gives rules for discovering solutions to problems. These are rules of the sort found in practical arts: general guidelines that make intuitive sense and seem to work well. They include, somewhat rephrased here, the following rules of problem solving. (If they seem too obvious to mention, I suggest you examine your own methods the next time you attempt to solve a problem; perhaps you're ignoring the "obvious"!)

- When considering different approaches to a problem, the less difficult precedes the more difficult, the more familiar precedes the less familiar, and an item with more points in common with the problem precedes one with fewer such points.
- When considering where to begin analyzing a problem, the whole precedes the parts, the principal parts precede other parts, the less remote parts precede more remote parts.
- When considering related problems, problems equivalent to the one being studied precede problems that are more or less ambitious, and these precede the rest. (Bilateral reduction precedes unilateral reduction, which precedes looser connections.) Formerly solved problems with the same kind of result as the problem you are trying to solve precede other formerly solved problems.

If it happens that you find yourself designing the paradigm requirement for a computer science department curriculum for the 1990s, I suggest that you consider putting mathematical discovery on the reading list.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 46.)*

```
PROGRAM QuittingTime;
{A rudimentary alarm/reminder program by Chris Derossi for Dr. Dobb's Journal. This simple
program shows how to write a faceless background task. You would use such a program as a
daemon, a background server, a spooler, or other tasks that need periodic time but don't
need to interact with the user.
  This particular example sits in the background waiting for a specific time to roll
around. Then it uses the Notification Manager to alert the user. }
USES
     Memtypes, OSIntf, ToolIntf, Notification;
CONST
     kIntResType      =     'INTV';
     kAlrtStrID       =     128;
     kAlrtTimeID      =     128;
TYPE
{These types are here so we can coerce pointers and handles to make Pascal happy. }
     NMRecPtr         =     ^NMRec;
     BooleanPtr       =     ^Boolean;
     IntPtr           =     ^Integer;
     IntHandle        =     ^IntPtr;
VAR
     doneFlag         :     Boolean;
     toldUser         :     Boolean;
     alarmTime        :     Integer;
     myNMBlock        :     NMRec;
     alrtStr          :     StringHandle;

PROCEDURE InitMyself;
{ Since this program doesn't have a user interface, it doesn't have to initialize the
Macintosh managers. All we do here is setup our global variables. }
BEGIN
     doneFlag         := FALSE;     {We just started!}
     toldUser         := FALSE;     {We haven't notified the user ever}
     alrtStr          := GetString(kAlrtStrID);
     HLock(Handle(alrtStr));
     alarmTime        := IntHandle(GetResource(kIntResType, kAlrtTimeID))^^;
END;

PROCEDURE AlertDone(nmBlockPtr : NMRecPtr);
{The user has dismissed the Notification Manager alert. This procedure removes the
Notification Manager queue element from the queue and sets the global variable doneFlag
to true. Since our A5 world is not guaranteed to be setup (and most likely isn't), we
use the pointer to the doneFlag variable stored in the NMRec nmRefCon field. }
VAR
     anErr     :     OSErr;
BEGIN
     BooleanPtr(nmBlockPtr^.nmRefCon)^ := TRUE;   {We can end the program now}
     anErr := NMRemove(QElemPtr(nmBlockPtr));
END;

PROCEDURE TellUser;
{If we haven't already told the user, then use the Notification Manager to display an alert. }
BEGIN
     IF NOT toldUser THEN BEGIN
          WITH myNMBlock DO BEGIN
               qType := nmType;      {MUST be nmType}
               nmMark := 0;          {No mark in the Apple Menu}
               nmSIcon := NIL;       {No Small Icon in the menu bar}
               nmSound := NIL;       {No beep}
               nmStr := Pointer(alrtStr^);{The string for the alert}
               nmResp := @AlertDone; {Our Notification Manager
                                      completion routine}
               nmRefCon := LongInt(@doneFlag);{Address of our global flag}
          END;
          IF NMInstall(@myNMBlock) = noErr THEN
               toldUser := TRUE;
     END;
END;

PROCEDURE MainLoop;
{ We'll call WaitNextEvent with a reasonable sleep value so we don't take too
much time from the rest of the system. Since we always run in the background,
we can never get anything but null events. }
CONST
     kSleepTime       =     60; {Get time once a second}
VAR
     anEvent          :     EventRecord;
     curTime          :     DateTimeRec;
     haveEvent        :     Boolean;
BEGIN
     haveEvent := WaitNextEvent(everyEvent, anEvent, kSleepTime, NIL);
     {Check to see if our alarm time has passed: }
     GetTime(curTime);
     IF (curTime.hour * 60 + curTime.minute) >= alarmTime THEN
          TellUser;
END;

BEGIN
     InitMyself;
     REPEAT
          MainLoop;
     UNTIL doneFlag;
END.
```

End Listing One

(continued on page 98)

Listing Two *(Text begins on page 46.)*

```
/*
 * QuittingTime - A rudimentary alarm/reminder program
 * by Chris Derossi for Dr. Dobb's Journal. MPW C Version.
 *
 * This simple program shows how to write a faceless background
 * task. You would use such a program as a daemon, a background server,
 * a spooler, or other tasks that need periodic time but don't need to
 * interact with the user.
 *
 * This particular example sits in the background waiting for a specific
 * time to roll around. Then it uses the Notification Manager to alert
 * the user.
 */


#include     <types.h>
#include     <memory.h>
#include     <events.h>
#include     <toolutils.h>
#include     <Resources.h>
#include     <Notification.h>

#define kIntResType   'INTV'
#define kAlrtStrID      128
#define kAlrtTimeID     128


Boolean               doneFlag;
Boolean               toldUser;
short                 alarmTime;
NMRec                 myNMBlock;
StringHandle          alrtStr;


void InitMyself()

/*Since this program doesn't have a user interface, it doesn't have to initialize
 *the Macintosh managers. All we do here is setup our global variables.
 */

{
     doneFlag = toldUser = false;
     alrtStr = GetString(kAlrtStrID);
     HLock(alrtStr);
     alarmTime = **(short **)(GetResource(kIntResType, kAlrtTimeID));
}


pascal void AlertDone(nmBlockPtr)

     NMRec          *nmBlockPtr;

/*
 *The user has dismissed the Notification Manager alert. This procedure
 *removes the Notification Manager queue element from the queue and
 *sets the global variable doneFlag to true. Since our A5 world is not
 *guaranteed to be setup (and most likely isn't), we use the pointer to
 *the doneFlag variable stored in the NMRec nmRefCon field.
 */

{
     OSErr          anErr;

     *(Boolean *)(nmBlockPtr->nmRefCon) = true; /* We can end the program now */
     anErr = NMRemove((QElemPtr)nmBlockPtr);
}


void TellUser()

/*
 *If we haven't already told the user, then use the Notification Manager
 * to display an alert.
 */

{
     if (!toldUser) {
          myNMBlock.qType = nmType;  /* MUST be nmType */
          myNMBlock.nmMark = 0;      /* No mark in the Apple Menu */
          myNMBlock.nmSIcon = nil;   /* No Small Icon in the menu bar */
          myNMBlock.nmSound = nil;   /* No beep */
          myNMBlock.nmStr = *alrtStr;/* The string for the alert */
          myNMBlock.nmResp = (ProcPtr)AlertDone;/* Our Notification Manager completion routine */
          myNMBlock.nmRefCon = &doneFlag;/* Address of our global flag */
          if (!NMInstall((QElemPtr)&myNMBlock))
               toldUser = true;
     }
}


void MainLoop()

/*
 * We'll call WaitNextEvent with a reasonable sleep value so we don't take too
 * much time from the rest of the system. Since we always run in the
 * background, we can never get anything but null events.
 */

#define     kSleepTime     60  /* Get time once a second */
{
     EventRecord          anEvent;
     DateTimeRec          curTime;

     WaitNextEvent(everyEvent, &anEvent, kSleepTime, nil);

     /* Check to see if our alarm time has passed: */
     GetTime(&curTime);
```

(continued on page 100)

MULTIFINDER

Listing Two *(Listing continued, text begins on page 46.)*

```
     if ((curTime.hour * 60 + curTime.minute) >= alarmTime)
          TellUser();
}


void main()
{
     InitMyself();
     while (!doneFlag)
          MainLoop();
}
```

End Listing Two

Listing Three

```
/*
 * QuttingTime.r - Rez source for background-only example program
 * by Chris Derossi
 */

 #include "Types.r"


type 'INTV' {  /* Resource type that contains a single integer value */
     integer;
};


resource 'INTV' (128) {
     17 * 60;  /* 1700 hrs or 5:00 pm */
};

resource 'STR ' (128) {
     "It's 5:00! Time to go home and relax!"
};

resource 'SIZE' (-1, purgeable) {
          dontSaveScreen,
          ignoreSuspendResumeEvents,
          enableOptionSwitch,
          canBackground,
          notMultiFinderAware,
          onlyBackground,
          dontGetFrontClicks,
          80 * 1024,
          80 * 1024
          };
```

End Listings

Listing One *(Text begins on page 62.)*

```
/*
Copyright (C) Magna Carta Software, 1988.  All Rights Reserved.
SMOOTH BROWSE -- A file browser to illustrate how to do smooth scrolling
and panning on the EGA and the VGA.
SYSTEM REQUIREMENTS:  EGA or VGA, Enhanced Color or Monochrome Display.
COMPILER SUPPORT: MSC 5.0+, Turbo C, WATCOM C, Mix Power C v1.1+.
Simple compiler invocation (suggested):
    MSC:            "CL /AS sb.c"
    TURBO:          "TCC sb.c" (from the environment).
    WATCOM C:       "WCC sb 3"
                    "WLINK FILE sb LIB ?:\watcomc\lib\clib%2, ?:\watcomc
                                                          \lib\math%2
                    OPTION Map, caseexact, stack=2048"
    MIX POWER C:    "PC /E sb.c"
Usage: UpArrow -- scroll up, DnArrow -- scroll down, + scroll faster,
- scroll slower, RtArrow -- smooth pan right, LtArrow -- smooth pan left
Space -- pause scrolling, PgUp, PgDn, Ctrl PgUp, Ctrl PgDn
First version: 3/5/88.  Last update 09-02-88. */
/* MSC SPECIFIC CODE */
#if !defined(__TURBOC__)    /* these two manifest constants are */
#if !defined(__POWERC)      /* #defined by the respective compilers.*/
#if !defined(__WATCOMC__)
                            /* Fall through... must be MSC! */
/* MSC _dos_findfirst structure compatible with TURBOC "findfirst" */
/* function */
struct find_t {
    char ff_reserved[21];
    char ff_attrib;
    unsigned ff_ftime;
    unsigned ff_fdate;
    long ff_fsize;
    char ff_name[13];
};
#define _FIND_T_DEFINED  /* A Microsoft manifest constant to prevent */
                         /* redef. */
#define ffblk find_t
#define findfirst(x,y,z)  _dos_findfirst(x,z,y)  /* note parm. order! */
#define bioskey(x)  _bios_keybrd(x)
#define inportb(port) inp(port)
#define outportb(port,value) outp((port),(value))
#define outport(port,value) outpw((port),(value))
#define MK_FP(seg,ofs)  ((void far *)((((unsigned long)
                                (seg)) << 16) | (ofs)))
#define peekb(a,b) (*((char far*)MK_FP((a),(b))))
#endif
#endif
#endif
/* WATCOM C SPECIFIC CODE */
#if defined(__WATCOMC__)
    #include <conio.h>
    #include <sys\types.h>
    #include <direct.h>

    #define inportb(port)           inp(port)
    #define outportb(port,value)    outp((port),(value))
    #define outport(port,value)     outpw((port),(value))
    #define peekb(a,b) (*((char far*)MK_FP((a),(b))))
    #define bioskey(x) kbhit()

    struct ffblk {
        char ff_reserved[21];
        char ff_attrib;
        unsigned ff_ftime;
        unsigned ff_fdate;
        long ff_fsize;
        char ff_name[13];
    };
    int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);

#endif
/* OTHER COMPILER-SPECIFIC CODE */
#if defined(__TURBOC__)
    #include <dir.h>
#elif !defined(__TURBOC__) && !defined(__WATCOMC__)
    #include <conio.h>
    #include <direct.h>
#endif

#include <dos.h>
#include <process.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char BYTE;     /* A convenient new data type */
/* MANIFEST CONSTANTS */
#define VERSION "1.0"
#define LOGICAL_WIDTH 132   /* EGA/VGA logical screen width (bytes, */
                            /* max. 512) */
#define MAXFILESIZE 0XFFFF  /* Maximum permisable file size to load */
#define NEWCOLOR    0X8     /* Code for new EGA color to load       */
#define BCOLOR      0       /* Screen background color              */
#define FCOLOR      15      /* Screen foreground color              */
#define TRUE        1
#define FALSE       !TRUE
#define TABSIZE     4
#define _KEYBRD_READY 1

#define KEYBOARD    0X16    /* The BIOS keyboard interrupt number */
#define VIDEO       0X10    /* The BIOS video interrupt number */
/* KEY DEFINITIONS */
#define ESC         1
#define UP_ARROW    72
#define DOWN_ARROW  80
#define LEFT_ARROW  75
#define RIGHT_ARROW 77
#define PAGE_UP     73
#define PAGE_DOWN   81
#define PLUS        78
#define MINUS       74
#define CPAGE_UP    132
#define CPAGE_DOWN  118
#define SPACEBAR    57
/* EGA AND VGA REGISTER VALUES */
#define PRESET_ROW_SCAN     8     /* Address of preset row scan reg. of */
                                  /* CRTC */
#define START_ADDRESS_HIGH  0X0C  /* Address of start address h reg. of */
                                  /* CRTC */
#define START_ADDRESS_LOW   0X0D  /* Address of start address l reg. of */
                                  /* CRTC */
#define AC_INDEX            0X3C0 /* Attribute Controller Index Register */
#define AC_HPP              0X13 | 0X20   /* Horizontal Pel Panning */
                                          /* Register */
#define AC_MCR              0X10  /* Attribute Controller Mode Control */
                                  /* Reg. */
#define LINE_COMPARE        0X18  /* CRTC line compare register. */
#define CRTC_OVERFLOW       0X07  /* CRTC overflow register. */
#define MAX_SCAN_LINE       0X09  /* CRTC maximum scan line register. */
/* GLOBAL VARIABLES */
struct ffblk u_file;          /* DOS file descriptor returned by */
                              /* findfirst() */
FILE *p_u_file;               /* Ptr. to the user file to browse */
unsigned end_of_file, screen_size, end_of_buffer, buffer_size;
unsigned right_edge, left_edge; /* TRUE if we are at the edge of the */
                                /* screen */
int vpel, hpel, mode, mono;
/* The following three structures are a convenient form in which to */
/* store video information and you can extend them with more */
/* information. We will declare one of each type (below). */

struct video_descriptor {
    unsigned ev_active;             /* if TRUE, an EGA or VGA is active */
    unsigned color;                 /* if TRUE, ega drives color monitor */
    unsigned ecd;                   /* if TRUE, Enhanced Color Display */
    unsigned ram;                   /* size of EGA memory (in k) */
    BYTE char_ht, char_wi; /* character height and width (EGA/VGA) */
    BYTE far *base;        /* starting address of the video buffer */
    unsigned address;               /* starting address of the active */
                                    /* video page */
    unsigned isr;                   /* EGA/VGA input status register */
                                    /* address */
    unsigned crtc;                  /* CRT Controller Register address */
};

struct screen_descriptor {
    unsigned rows;
    unsigned cols;
    unsigned a_start;       /* start address of screen A in split screen */
                            /* mode */
    unsigned logical_width;     /* screen logical width in words. */
                                /* max: 0x100 */
    unsigned start_addr;        /* screen start address. max: 0X4000 */
};

struct enhanced_graphics_adapter {
    unsigned present;               /* if TRUE, EGA present */
    unsigned active;                /* if TRUE, EGA active */
};

struct video_graphics_array {
    unsigned present;               /* if TRUE, VGA present */
    unsigned active;                /* if TRUE, EGA present */

};

struct video_descriptor video;
struct screen_descriptor screen;
struct enhanced_graphics_adapter ega;
struct video_graphics_array vga;
/* FUNCTION PROTOTYPES -- In order of appearance */
int             main_menu(BYTE *p_fbuf);
void            intro_to_buffer(unsigned state);
BYTE * build_line(BYTE **p_fbuf);
int             line_to_buffer(BYTE *p_fbuf);
void            smooth_scroll(int count, unsigned speed);
int             smooth_pan(int count, unsigned speed);
unsigned        set_logical_screen_width(unsigned l_width);
void            set_start_addr(unsigned start_addr);
void            set_pel_pan(int hpel);
void            set_line_compare(unsigned scan_line);
void            load_ega_color(BYTE pregister, BYTE color);
unsigned        set_video_mode(BYTE m);
int             get_v_mode(void);
void            get_v_config(void);
unsigned        get_key(void);
int             get_cursor_size(void);
void            set_cursor_size(BYTE top_line, BYTE bottom_line);
void            error(char *fs,...);

int main(int argc, char *argv[])
{

    BYTE *p_fbuf;
    int not_found, rc;
    int csize;
    /* TEST FOR A FILENAME/PATH */
    /* No filename entered -- give syntax message and exit */
    if(argc < 2) {
       fprintf(stderr,"Smooth browser version %s by\n\tMagna Carta Software
                \n\tP.O. Box 475594\n\tGarland, Texas 75047\n", VERSION);
        fprintf(stderr, "Usage: sb pathname\n");
        exit(1);
    }
    /* SEE IF FILE EXISTS */
```

(continued on next page)

Listing One *(Listing continued, text begins on page 62.)*

```
  not_found = findfirst(argv[1],&u_file,0);
  if (not_found) error("\nFile %s not found.",argv[1]);
  /* get file size. If too large (> MAXFILESIZE) then tell the user */
  /* and exit */
  if (u_file.ff_fsize > MAXFILESIZE)
      error("\nMaximum file size %u bytes exceeded\n",MAXFILESIZE);
  /* TEST FOR AN ACCEPTABLE VIDEO CONFIGURATION */
  mode = get_v_mode();
  get_v_config();
  if (!vga.present && !ega.present)
      error("EGA/VGA required to run SMOOTH BROWSE");
  if (!video.ev_active)
      error("The EGA/VGA must be active to use SMOOTH BROWSE");
  if (!video.ecd && video.color)
      error("Enhanced Color or Monochrome Display required to run
            SMOOTH BROWSE");
  if (video.ram < 128)
      error("More video RAM required to use SMOOTH BROWSE");
  /* CREATE BUFFER FOR FILE. IF ERROR, RETURN AN ERRORLEVEL TO DOS */
  p_fbuf = (BYTE *) calloc(1, (unsigned) u_file.ff_fsize);
  if (p_fbuf == NULL) error("Not enough memory available to load file");
  /* OPEN FILE. IF THERE IS AN ERROR RETURN AN ERRORLEVEL TO DOS */
  p_u_file = fopen(argv[1], "rt");
  if (p_u_file == NULL) error("Error opening file %s",u_file.ff_name);
  /* READ IN FILE. IF THERE IS A SHORT COUNT RETURN AN ERRORLEVEL TO DOS */
  if (fread(p_fbuf,1,(int) u_file.ff_fsize,p_u_file) == NULL)
      error("Short count returned when reading file");
  /* SYSTEM CHECKS OUT AND FILE EXISTS AND IS WITHIN SIZE LIMITS */
  /* NOTE: VARIABLE MONO IS SET BY THE FUNCTION GET_V_MODE() */
  if (mode != 7) set_video_mode(3);
  else set_video_mode(7);
  /* GET THE ADDRESS OF THE INPUT STATUS REGISTER AND CRT CONTROLLER */
  if (mono) video.isr = 0x03ba;
  else video.isr = 0x03da;
  video.crtc = video.isr - 6;
  /* SAVE THE CURSOR AND TURN IT OFF */
  csize = get_cursor_size();
  set_cursor_size(0,0);
  /* Load a pleasant but unusual EGA color */
  load_ega_color(0,NEWCOLOR);
  /* DO IT */
  rc = main_menu(p_fbuf);
  /* EXIT ROUTINES -- RESTORE THE CURSOR AND RESET THE VIDEO MODE. */
  set_cursor_size((BYTE) (csize >> 8), (BYTE) csize);
  if (mono) set_video_mode(7);
  else set_video_mode(3);
  return(rc);
}

int main_menu(BYTE *p_fbuf)
{
  unsigned speed=1;
  int v_direction=0, old_v_direction=0;
  int h_direction=0, old_h_direction=0;
  unsigned scan;
  BYTE old_reg;
  int ret;
  /* SET LOGICAL SCREEN WIDTH */
  screen.logical_width = LOGICAL_WIDTH;
  set_logical_screen_width(screen.logical_width);
  /* SPLIT THE SCREEN */
  screen.a_start = screen.logical_width;
  set_line_compare((screen.rows - 1) * video.char_ht);
  /* CREATE 16 VIDEO PAGES IN VIDEO MODE 3 -- MANY CAVEATS TO THIS */
  outportb(0X3CE, 6);                    /* Point index at Misc. register */
  if (vga.active) old_reg = (BYTE) inportb(0X3CF); /* save contents */
  else old_reg = 0X0E;
  outportb(0X3CF, old_reg & 0XF7);    /* reset EGA ram to A0000-AFFFF */
  video.base = (BYTE far *) 0XA0000000L;
  /* SET SCREEN START ADDRESS FOR FILE */
  screen.start_addr = screen.a_start;
  set_start_addr(screen.start_addr);
  screen_size = screen.logical_width*(screen.rows-1);
  /* DISPLAY THE INTRODUCTORY MESSAGE */
  intro_to_buffer(TRUE);
  /* DISPLAY THE FILE */
  line_to_buffer(p_fbuf);              /* position file on screen */
  end_of_buffer = end_of_file/2 - screen_size;
  inportb(video.isr); /* a dummy read to initialize the Attribute */
                      /* Controller */

  left_edge = TRUE;   /* file display starts at left edge */
  /* THIS IS THE MAIN LOOP THAT MOVES LOGICAL SCREEN 'A' AROUND */
  do {
      do {
          if (v_direction) {
              smooth_scroll(v_direction,speed);
              old_v_direction = old_h_direction = 0;
          }
          else if (h_direction) {
              ret = smooth_pan(h_direction,speed);
              if (ret == -1) h_direction = v_direction = 0;
              old_h_direction = old_v_direction = 0;
          }
      } while (!kbhit());

      scan = get_key();
      switch(scan) {
          case  ESC:      /* The user's key presses tell us what to do */
              break;
          case UP_ARROW:
              v_direction = -2;
              h_direction = 0;
              break;
          case DOWN_ARROW:
              v_direction = 2;
              h_direction = 0;
              break;
          case LEFT_ARROW:
              intro_to_buffer(FALSE);
              h_direction = -1;
              v_direction = 0;
              break;
          case RIGHT_ARROW:
              intro_to_buffer(FALSE);
              h_direction = 1;
              v_direction = 0;
              break;
          case PAGE_UP:
              if (screen.start_addr > screen_size + screen.logical_
                  width) screen.start_addr -= screen_size;
              else {
                  screen.start_addr = screen.a_start;
                  left_edge = TRUE;
              }
              vpel = 0;       /* Set the preset row scan register to 0 */
              /* address the preset row scan register */
              outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
              set_start_addr(screen.start_addr);
              break;
          case PAGE_DOWN:                                  /* pg dn */
              if (screen.start_addr < end_of_buffer - screen_size)
                  screen.start_addr += screen_size;
              else screen.start_addr = end_of_buffer;
              vpel = 0;       /* Set the preset row scan register to 0 */
              outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
              /* reset pel panning position */
              hpel = (mono || vga.active) ? 8 : 0;
              set_pel_pan(hpel);
              set_start_addr(screen.start_addr);
              break;
          case PLUS:                       /* '+' key -- scroll faster */
              if (speed < 5) speed++;
              break;
          case MINUS:                      /* '-' key -- scroll slower */
              if (speed > 1) speed--;
              break;
          case SPACEBAR:                /* space bar -- toggle scrolling */
              if (v_direction) {        /* moving vertically, so stop */
                  old_v_direction = v_direction;
                  v_direction = 0;
              }
              else if (h_direction) {   /* moving horizontally, so stop */
                  old_h_direction = h_direction;
                  h_direction = 0;
              }
              else if (old_v_direction) {  /* stopped -- start vertical */
                  v_direction = old_v_direction;
              }
              else if (old_h_direction) {  /* stopped -- start */
                                           /* horizontal */
                  h_direction = old_h_direction;
              }
              break;
          case CPAGE_UP:             /* ctrl-pgup -- go to top */
              screen.start_addr = screen.a_start;
              left_edge = TRUE;
              /* reset pel panning position */
              vpel = 0; /* Set the preset row scan register to 0 */
              outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
              hpel = (mono || vga.active) ? 8 : 0;
              set_start_addr(screen.start_addr);
              set_pel_pan(hpel);
              break;
            case CPAGE_DOWN:    /* ctrl-pgdn -- go to bottom */
                screen.start_addr = end_of_buffer;
                /* reset pel panning position */
                vpel = 0;   /* Set the preset row scan register to 0 */
                outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
                hpel = (mono || vga.active) ? 8 : 0;
                set_pel_pan(hpel);

                set_start_addr(screen.start_addr);
                break;

            default:
                break;
      }
  } while (scan != 1);

  return (0);
}
/* INTRO_TO_BUFFER -- Writes the introductory messsage on screen 'B' */
void intro_to_buffer(unsigned state)
{
    BYTE far *ega_buf ;
    static BYTE a_sbuf[] = "SMOOTH BROWSE v1.0 -- by Magna Carta
                            Software, 1988";
    BYTE *p_sbuf, att;

    ega_buf = video.base;
    att = ((FCOLOR & 0X7) << 4) + BCOLOR;
    if (state) {
        p_sbuf = a_sbuf;
        while (*p_sbuf) {
            *ega_buf++ = *p_sbuf++;
            *ega_buf++ = att;
```

(continued on page 104)

Listing One *(Listing continued, text begins on page 62.)*

```
        }
    }
    while (ega_buf < video.base + (screen.logical_width << 1) ) {
        *ega_buf++ = '\x20';
        *ega_buf++ = att;
    }
}
/* BUILD_LINE -- This function constructs a line of text from data read */
/* in from the file. */
BYTE *build_line(BYTE **p_fbuf)
{
    static BYTE lbuf[LOGICAL_WIDTH + 1];/* add 1 for '\0' terminator */
    BYTE *p_lbuf;
    int i;

    p_lbuf = lbuf;  /* point to the buffer to hold the line that we */
                    /* build */
    /* construct a line in the buffer for display until we hit a */
    /* '\n' or '\t'*/
    while (**p_fbuf != '\n') {
        if (**p_fbuf == '\t') for(i=0;i<TABSIZE;i++) *p_lbuf++ = 0x20;
        else if (**p_fbuf >= 0x20 && **p_fbuf < 0x80) *p_lbuf++ =
                                                        **p_fbuf;
        (*p_fbuf)++;
        if (p_lbuf >= lbuf + screen.logical_width) break;
    }
    *p_lbuf = '\0';
    (*p_fbuf)++;        /* advance past the new line character */

    return (lbuf);
}
/* LINE_TO_BUFFER -- Moves the line from memory to the video buffer */
int line_to_buffer(BYTE *p_fbuf)
{
    BYTE far *p_vbuff, far *line, far *eob;
    BYTE *p_lbuf, att;
    unsigned i;

    p_vbuff = video.base + 2*screen.logical_width;
    att = (BCOLOR << 4) + FCOLOR;
    for (i = 0; i < 0X8000 - 2*screen.logical_width; i++) {
        p_vbuff[i++] = '\0';
        p_vbuff[i] = att;
    }
    eob = video.base + (0XFFFF - screen.logical_width);
    line = p_vbuff;
    while (*p_fbuf && p_vbuff < eob ) {
        p_lbuf = build_line(&p_fbuf);
        while (*p_lbuf) {
            *p_vbuff++ = *p_lbuf++;
            *p_vbuff++ = att;
        }
        /* go to next screen line */
        line += screen.logical_width*2;
        p_vbuff = line;
    }
    end_of_file = FP_OFF(p_vbuff);
    return (0);
}
/* SMOOTH_SCROLL scrolls the EGA video buffer the number of scan lines
indicated in "count" at a speed of "speed" scan lines per vertical
retrace. One retrace occurs each 60th of a second regardless of
processor speed.
*/
void smooth_scroll(int count, unsigned speed)
{
      unsigned i;
      /* GET THE START ADDRESS OF THE SCREEN BUFFER */
      outportb(video.crtc, START_ADDRESS_HIGH);          /* High byte */
      screen.start_addr = inportb(video.crtc+1) << 8;
      outportb(video.crtc, START_ADDRESS_LOW);           /* Low byte */
      screen.start_addr |= inportb(video.crtc+1);
      /* count > 0 => scroll screen upwards. */
      /* i is the number of scan lines scrolled */
      if (count>0 && (screen.start_addr < end_of_buffer))
      for(i=0;i < count;) {
          if (vpel >= (int) video.char_ht) {
              vpel = vpel - video.char_ht;
              screen.start_addr += screen.logical_width;
          }
          if (vpel < 0) {
              if (screen.start_addr > screen.logical_width) {
                  vpel += video.char_ht;
                  screen.start_addr -= screen.logical_width;
              }
              else vpel = 0;
          }
          for(;vpel< (int) video.char_ht && i<count;vpel+=speed,i+=speed) {
              /* wait for a vertical retrace */
              while (!(inportb(video.isr) & 8));
              /* wait for horizontal or vertical retrace */
              while (inportb(video.isr) & 1);
              /* address the preset row scan register */
              outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
              /* Reset the start address */
              set_start_addr(screen.start_addr);
          }
      }
      /* count < 0  => scroll screen characters downwards */
      /* i is the number of scan lines scrolled */
      if (count < 0) {
          if (vpel >= (int) video.char_ht) vpel -= video.char_ht;

          for(i=0;i < -count && screen.start_addr;) {
          /* This loop determines whether to update the start address */
          /* It is iterated once for each video.char_ht. */
              if (vpel < 0) {
                  vpel = video.char_ht + vpel;
                  if (screen.start_addr > screen.logical_width)
                      screen.start_addr -= screen.logical_width;
                  else {
                      screen.start_addr = screen.a_start;
                      vpel=0;
                  }
              }
              /* this loop moves the screen "speed" scan lines each time */
              /* through */
              for(;vpel>=0 && i< -count;vpel-=speed,i+=speed) {
                  /* wait for a vertical retrace */
                  while (!(inportb(video.isr) & 8));
                  /* wait for horizontal or vertical retrace */
                  while (inportb(video.isr) & 1);
                  /* address the preset row scan register */
                  outport(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
                  /* Reset the start address */
                  set_start_addr(screen.start_addr);
              }
          }
      }
}
/* SMOOTH_PAN
This function invokes smooth panning on the EGA/VGA in text mode.  The
function calculates the number of scan lines per row.  The speed variable
adjusts the speed in pixels per vertical retrace and takes values in the
range 1-8 for EGA color and 1-9 for monochrome and the VGA.
*/
int smooth_pan(int count, unsigned speed)
{
    unsigned i;
    /* count greater than zero (move viewport to the right) */
    if (count>0 && !right_edge) for(i=0;i<count;) {
        /* if we have scrolled a full character, reset start address */
        if (hpel >= 8) {
            screen.start_addr++;
            set_start_addr(screen.start_addr);
            if (!left_edge) if (!(screen.start_addr % (screen.logical_
                                                        width)))
                right_edge = TRUE;
            left_edge = FALSE;
            if (!right_edge) {
                if (!(mono || vga.active)) hpel %= 8; /* reset the */
                                                      /*  pel counter */
                else {
                    hpel %= 9;
                    if (hpel == 8) {
                        set_pel_pan(hpel);
                        hpel += speed;
                        hpel %= 9;
                    }
                }
            }
            else {
                hpel = (mono || vga.active) ? 8 : 0;
                set_pel_pan(hpel);
                return (-1);
            }
        }
            for(;hpel < 8 && i < count;i+=speed, hpel+=speed) set_pel_
                                                            pan(hpel);
        }
        /* count less than zero (move viewport to the left) */
        else if (count < 0) {
            if (mono || vga.active) {
                if (left_edge && hpel == 8) return (-1);
                if (hpel > 8) hpel -= 9;
            }
            else {
                if (left_edge && hpel == 0) return (-1);
                if (hpel > 7) hpel = hpel - 8;
            }
            for(i=0; i < (-count); ) {
            /* IF WE HAVE SCROLLED A FULL CHARACTER, RESET START ADDRESS */
                if (hpel<0 && !left_edge) {
                    if ((mono || vga.active) && hpel == -1) {
                        hpel = 8;
                        set_pel_pan(hpel);
                        hpel = -1 - speed;
                    }
                    screen.start_addr--;
                    right_edge = FALSE;
                    hpel = (mono || vga.active) ? 9 + hpel : 8 + hpel;
                    set_start_addr(screen.start_addr);
                    if (!(screen.start_addr % screen.logical_width)) {
                        left_edge = TRUE;
                        screen.start_addr--;
                    }
                }
                else if (hpel<0 && left_edge) i = (-count);

                for(;hpel >= 0 && i < (-count);i+=speed, hpel-=speed) set_
                                                            pel_pan(hpel);
                if (left_edge && hpel < speed) {
                    hpel = (mono || vga.active) ? 8 : 0;
                    set_pel_pan(hpel);
                    return (-1);
                }
            }
        }
        return (0);
    }
    /* SET_LOGICAL_SCREEN_WIDTH defines an EGA/VGA screen width for smooth
    panning and sets the the global variable "screen.cols."  screen.cols
    must be restored when smooth panning is finished or subsequent screen
    writes will address the wrong screen locations.
    l_width is the width of the logical screen in bytes. Max 512.
    */
```

(continued on page 106)

Listing One *(Listing continued, text begins on page 62.)*

```
unsigned set_logical_screen_width(unsigned l_width)
{
    l_width = (l_width > 512) ? 512 : l_width;
    /* set screen_cols */
    screen.cols = l_width;
    /* set logical screen width */
    l_width >>= 1;                          /* convert to words */
    outport(video.crtc, (l_width << 8) | 0x013);

    return (l_width << 1);
}
/* SET_START_ADDRESS -- Sets the start address of the screen. I.e. */
/* the address that occupies the top left of the screen. */
void set_start_addr(unsigned start_addr)
{
    /* address the start address high register */
    outport(video.crtc, (start_addr & 0XFF00) | START_ADDRESS_HIGH);
    /* address the start address low register */
    outport(video.crtc, (start_addr << 8) | START_ADDRESS_LOW);
}
/* SET_PEL_PAN -- Sets the horizontal pel panning register to the */
/* value specified in "hpel".  Called by smooth_pan() */
void set_pel_pan(int hpel)
{
    /* first wait for end of vertical retrace */
    while (inportb(video.isr) & 8);
    /* wait for next vertical retrace */
    while (!(inportb(video.isr) & 8));
    /* address the horizontal pel paning register */
    outportb(AC_INDEX, AC_HPP);
    outportb(AC_INDEX, hpel);
}
/* SET_LINE_COMPARE -- Sets the scan line at which the screen is split.*/
void set_line_compare(unsigned scan_line)
{
    BYTE old_reg;
    if (scan_line > 256) {              /* set bit 4 of the overflow */
                                        /* register */
        if (ega.active) {
            if ((mode >= 4 && mode <= 7) || (mode == 0X0D || mode ==
                                                        0X0E))
                outport(video.crtc, (0X11 << 8) | CRTC_OVERFLOW);
            else outport(video.crtc, (0X1F << 8) | CRTC_OVERFLOW);
            if (mode < 4 && !video.ecd) outport(video.crtc,
                                    (0X11 << 8) | CRTC_OVERFLOW);
        }
        else if (vga.active) {      /* ahhh... readable registers */
            /* set bit 8 of line compare reg., which is bit 4 of */
            /* the CRTC overflow reg. */
            outportb(video.crtc,CRTC_OVERFLOW);
            old_reg = (BYTE) inportb(video.crtc+1);
            outportb(video.crtc+1,old_reg | 0X10);
            /* set bit 9 of line compare reg., which is bit 6 of
            the CRTC max. scan line register */

            outportb(video.crtc,MAX_SCAN_LINE);
            old_reg = (BYTE) inportb(video.crtc+1);
            outportb(video.crtc+1,old_reg & 0XBF);
        }
        scan_line -= 256;
    }
    else {
        if (ega.active) {
            /* clear bit 8 of the line compare register (which is */
            /* stored as bit 4 of the CRTC overflow register).  This */
            /* register is write only.  */
            if (!video.ecd && mode < 4) outport(video.crtc,
                                    (1 << 8) | CRTC_OVERFLOW);
            else if (video.ecd && mode < 4) outport(video.crtc,
                                     (0XF << 8) | CRTC_OVERFLOW);
            if ((mode >= 4 && mode < 7) || (mode == 0XD || mode == 0XE)
                    ) outport(video.crtc, (1 << 8) | CRTC_OVERFLOW)8;
            if (mode == 7 || mode > 0X0E) outport(video.crtc,
                                    (0XF << 8) | CRTC_OVERFLOW);
        }
        if (vga.active) {
            outportb(video.crtc,CRTC_OVERFLOW); /* clear bit 8 of line */
                                                /* compare reg. */
            old_reg = (BYTE) inportb(video.crtc+1);
            outportb(video.crtc,((old_reg & 0XEF) << 8) | CRTC_OVERFLOW);
            outportb(video.crtc,MAX_SCAN_LINE);  /* clear bit 9 of line */
                                                 /* compare reg. */
            old_reg = (BYTE) inportb(video.crtc+1);
            outportb(video.crtc,((old_reg & 0XBF) << 8) | MAX_SCAN_LINE);
        }
    }
    outport(video.crtc, (scan_line << 8) | LINE_COMPARE);
}
/********************** UTILITY FUNCTIONS ****************************/
/* LOAD_EGA_COLOR -- Load an EGA or VGA palette register with a */
/* selected color */
void load_ega_color(BYTE pregister, BYTE color)
{
    union REGS regs;

    regs.x.ax = 0x1000;     /* Service 0. Set an individual palette */
                            /* register */
    regs.h.bl = pregister;  /* BL holds the register to be set (0-15) */
    regs.h.bh = color;      /* BH hols the new color */
    int86(VIDEO, &regs, &regs);
}
/* SET_VIDEO_MODE -- sets the active video mode. */
unsigned set_video_mode(BYTE m)
{
    union REGS regs;

    regs.h.ah = 0;          /* service 0 - set current video mode */
    regs.h.al = m;          /* requested mode # */
    int86(VIDEO, &regs, &regs);

    return m;
}
/* GET_V_MODE -- Returns the video mode and sets the global variables */
/* screen.cols (# of screen columns), screen.rows (# of screen rows)  */
/* video.base (screen buffer starting address), and mono.             */
int get_v_mode(void)
{
    union REGS regs;

    regs.h.ah = 0xf;        /* get current video mode */
    int86(VIDEO, &regs, &regs);
    mode = regs.h.al;       /* current mode # */
    screen.cols = regs.h.ah;
    regs.h.dl = 0x18;       /* dummy #rows for CGA compatibility */
    regs.h.bh = 0;
    regs.x.ax = 0x1130;     /* BIOS function 0x11, sub-function 0x30 */
    int86(VIDEO,&regs,&regs);
    screen.rows = regs.h.dl + 1;/* screen rows returned in DL */
    if (mode == 7) {
        video.base = (BYTE far *) 0XB0000000L;
        mono = TRUE;
    }
    else {
        video.base = (BYTE far *) 0XB8000000L;
        mono = FALSE;
    }

    return (mode);
}
/* GET_V_CONFIG -- A portmanteau routine to determine if an EGA,  */
/* VGA or neither is present and, if an EGA or VGA is present,    */
/* to set the following video parameters:                         */
/* vga.present = TRUE if a VGA is present in the system.          */
/* vga.active = TRUE if the VGA is the active adapter.            */
/* ega.present = TRUE if an EGA is present in the system.         */
/* ega.active = TRUE if an EGA is the active adapter.             */
/* video.ram = amount of video RAM installed on the adapter.      */
/* video.color = TRUE if the EGA/VGA drives a color display.      */
/* video.ecd = TRUE if the EGA/VGA drives an enhanced display.    */
/* video.ev_active = TRUE if the EGA or the VGA is active.        */
/* This function should be preceeded by a call to get_v_mode().   */
void get_v_config(void)
{
    union REGS regs;
    BYTE e_byte;            /* EGA information byte in ROM data area */
    /* First, check for the presence of an VGA */
    regs.x.ax = 0x1a00; /* Function call 0x1a -- Tech. Ref./2 p. 3-24 */
    int86(VIDEO, &regs, &regs);         /* Function supported => VGA */
    if (regs.h.al == 0x1a) {
        vga.present = TRUE;
        if (regs.h.bl < 7) vga.active = FALSE;
        else vga.active = TRUE;
    }
    else {
        vga.present = FALSE;
        vga.active = FALSE;
    }
    /* Check for the presence of an EGA and set relevant EGA/VGA */
    /* parameters */
    regs.h.ah = 0x12; /* EGA BIOS alternate select. Tech. Ref. p106. */
    regs.h.bl = 0x10; /* return EGA information.                  */
    int86(VIDEO, &regs, &regs);
    if (regs.h.bl == 0x10 || vga.present) ega.present = FALSE;
    else ega.present = TRUE;
    /* If an EGA or VGA is present then set some important parameters */
    if (ega.present || vga.present) {   /* EGA/VGA is present -- is */
                                        /* it active? */
        video.ram = regs.h.bl*64 + 64;  /* EGA/VGA installed memory */
        if (regs.h.bh) video.color = FALSE; /* EGA/VGA drives a mono */
                                            /* monitor */
        else video.color = TRUE;            /* EGA/VGA drives a color */
                                            /* monitor */
        /* See if EGA/VGA drives an Enhanced Color Display */
        if (video.color) {
            if (regs.h.cl == 3 || regs.h.cl == 9) video.ecd = TRUE;
            else video.ecd = FALSE;
        }
        e_byte = peekb(0,0x487);                    /* EGA info. byte */
        if (e_byte & 8) video.ev_active = FALSE;    /* EGA not active */
        else {
            video.ev_active = TRUE;                 /* EGA is active */
            if (!vga.active) ega.active = TRUE;
        }
    }

    if (vga.active) {
        regs.x.ax = 0X1202;
        regs.h.bl = 0X30;
        int86(VIDEO, &regs, &regs);
    }
    /* check # of screen rows for font size */
    if (vga.active) video.char_ht = (BYTE) (400/screen.rows);
    else if (ega.active) if (video.ecd || mono) video.char_ht =
                                                (BYTE) (350/screen.rows);
    else video.char_ht = (BYTE) (200/screen.rows);
}
/* GET_KEY -- Returns the scan code of the next key in the keyboard */
/* buffer. */
unsigned get_key(void)
{
    union REGS regs;

    regs.h.ah = 0;      /* read next keyboard character */
```

(continued on page 108)

Listing One *(Listing continued, text begins on page 62.)*

```
    int86(KEYBOARD, &regs, &regs);
    return ((unsigned) regs.h.ah);
}
/* GET_CURSOR_ SIZE -- Get the cursor size on the active page. */
int get_cursor_size(void)
{
    union REGS regs;
    int hi,lo;

    regs.h.ah = 3;                  /* request cursor size */
    regs.h.bh = 0;                  /* from page 0 */
    int86(VIDEO, &regs, &regs);     /* ROM BIOS video service */
    hi = regs.h.ch;                 /* top scan line of cursor */
    lo = regs.h.cl;                 /* bottom scan line of cursor */
    return (hi << 8) | lo;          /* combine for return */
}
/* SET CURSOR SIZE sets the cursor size. top_line=bottom_line=0 turns */
/* it off  */
void set_cursor_size(BYTE top_line, BYTE bottom_line)
{
    union REGS regs;

    regs.h.ah = 1;              /* BIOS function 1, set cursor size */
    if (top_line == 0 && bottom_line == 0) regs.h.ch = 32; /* request */
                                                          /* cursor off */
    else {
        regs.h.ch = top_line;               /* row */
        regs.h.cl = bottom_line;            /* column */
    }
    int86(VIDEO, &regs, &regs);
}
/* ERROR -- A simple error-handler */
void error(char *fs,...)
{
    va_list argptr;
    va_start(argptr,fs);
    vfprintf(stderr,fs,argptr);
    va_end(argptr);
    exit(1);
}
 #if defined(__WATCOMC__)
 /*
 WATCOM C does not access the DOS file system in a fashion similar to
 the other compilers so we write our own "findfirst()" function.
 WATCOM C uses a function called opendir(), which could be used instead.
 */
 /* NOTE: THE FOLLOWING IMPLEMENTATION IS VALID ONLY IN SMALL */
 /* DATA MODELS */
 int findfirst(const char *pathname, struct ffblk *ffblk, int attrib)
 {
     union REGS regs;
     struct SREGS sregs;

     segread(&sregs);
     regs.h.ah = 0X1A;              /* DOS set DTA function         */
     regs.x.dx = (unsigned) ffblk;  /* address of new DTA address   */
     intdosx(&regs, &regs, &sregs); /* this reads the registers...  */

     regs.h.ah = 0X4E;              /* DOS find first matching file */
     regs.x.cx = attrib;            /* search attribute             */
     regs.x.dx = (unsigned) pathname;/* DS:DX is address of pathname */
     intdosx(&regs, &regs, &sregs);
     return (regs.x.ax);
 }
 #endif
```

End Listing One

ListingTwo

```
/*
SMOOTHLIB--A library of C language routines to do smooth scrolling and
panning on the EGA and the VGA. by Andrew J. Chalk.
SYSTEM REQUIREMENTS: EGA or VGA. Enhanced Color, Monochrome or Analog
                     Display.
COMPILER SUPPORT: MSC 5.0+/Quick C
Simple compiler invocation (suggested):
    MSC:        "CL /AS sb.c"
First version: 3/5/88.  Last update 09-01-88.
*/

#include <conio.h>
#include <stdio.h>

typedef unsigned char BYTE;     /* A convenient new data type */
/* MANIFEST CONSTANTS */
#define LOGICAL_WIDTH 132  /* EGA/VGA logical screen width (bytes, */
                           /* max. 512) */
#define TRUE            1
#define FALSE       !TRUE
/* Macros to make a far pointer and examine at a byte in memory */
#define MK_FP(seg,ofs)  ((void far *)((((unsigned long)(seg)) << 16) |
                                                              (ofs)))
#define peekb(a,b)      (*((char far*)MK_FP((a),(b))))
/* EGA AND VGA REGISTER VALUES */
#define PRESET_ROW_SCAN     8     /* Address of preset row scan reg. */
                                  /* of CRTC */
#define START_ADDRESS_HIGH  0X0C  /* Address of start address h reg. */
                                  /* of CRTC */
#define START_ADDRESS_LOW   0X0D  /* Address of start address l reg. */
                                  /* of CRTC */
#define AC_INDEX            0X3C0 /* Attribute Controller Index Register
#define AC_HPP              0X13 | 0X20 /* Horizontal Pel Panning Register
#define AC_MCR              0X10  /* Attribute Controller Mode */
                                  /* Control Reg.  */
#define LINE_COMPARE        0X18  /* CRTC line compare register.*/
#define CRTC_OVERFLOW       0X07  /* CRTC overflow register.          */
#define MAX_SCAN_LINE       0X09  /* CRTC maximum scan line register.  */
/* GLOBAL VARIABLES */
unsigned end_of_buffer;      /* address of the end of the video buffer */
unsigned right_edge, left_edge; /* TRUE if we are at the edge of the */
                                /* screen */
int vpel, hpel;                 /* vertical pel height, etc.         */
int mode, mono;              /* video mode, monochrome (TRUE or FALSE) */
/* STRUCTURE DEFINITIONS
The following four structures are a convenient form in which to store
video information and can be extended with additional members.
We will declare one of each type (below).
*/
struct video_descriptor {
    unsigned ev_active;    /* if TRUE, an EGA or VGA is active     */
    unsigned color;        /* if TRUE, ega drives color monitor    */
    unsigned ecd;          /* if TRUE, Enhanced Color Display      */
    unsigned ram;          /* size of EGA memory (in k)            */
    BYTE char_ht, char_wi; /* character height and width (EGA/VGA) */
    BYTE far *base;        /* start address of the video buffer    */
    unsigned address;      /* start addr. of the active video page */
    unsigned isr;          /* EGA/VGA input status register address*/
    unsigned crtc;         /* CRT Controller Register address      */
};
/* We place all the information related to the screen in one structure */
struct screen_descriptor {
    unsigned rows;
    unsigned cols;
    unsigned a_start; /* start address of screen A in split screen mode */
    unsigned logical_width; /* screen logical width in words. max: 0x100 */
    unsigned start_addr; /* screen start address. max: 0X4000 */
};
/* This structure contains information specific to the EGA */
struct enhanced_graphics_adapter {
    unsigned present;               /* if TRUE, EGA present */
    unsigned active;                /* if TRUE, EGA active */
};
/* This structure contains information specific to the VGA */
struct video_graphics_array {
    unsigned present;               /* if TRUE, VGA present */
    unsigned active;                /* if TRUE, EGA present */

};
/* STRUCTURE DECLARATIONS */
struct video_descriptor video;
struct screen_descriptor screen;
struct enhanced_graphics_adapter ega;
struct video_graphics_array vga;
/* FUNCTION PROTOTYPES -- In order of appearance */
void        smooth_scroll(int count, unsigned speed);
int         smooth_pan(int count, unsigned speed);
unsigned    set_logical_screen_width(unsigned l_width);
void        set_start_addr(unsigned start_addr);
void        set_pel_pan(int hpel);
/*** FUNCTION DEFINITIONS BEGIN HERE ***/
/*
SMOOTH_SCROLL scrolls the EGA/VGA video buffer in text mode the number of
scan lines indicated in "count" at a speed of "speed" scan lines per
vertical retrace.
*/
void smooth_scroll(int count, unsigned speed)
{
    unsigned i;
    /* GET THE START ADDRESS OF THE SCREEN BUFFER */
    outp(video.crtc, START_ADDRESS_HIGH);       /* High byte */
    screen.start_addr = inp(video.crtc+1) << 8;
    outp(video.crtc, START_ADDRESS_LOW);        /* Low byte */
    screen.start_addr |= inp(video.crtc+1);
    /* COUNT > 0 => SCROLL SCREEN UPWARDS. */
    /* i is the number of scan lines scrolled */
    if (count>0 && (screen.start_addr < end_of_buffer))
    for(i=0;i < count;) {
        if (vpel >= (int) video.char_ht) {
            vpel = vpel - video.char_ht;
            screen.start_addr += screen.logical_width;
        }
        if (vpel < 0) {
            if (screen.start_addr > screen.logical_width) {
                vpel += video.char_ht;
                screen.start_addr -= screen.logical_width;
            }
            else vpel = 0;
        }
        for(;vpel< (int) video.char_ht && i<count;vpel+=speed,i+=speed) {
            /* wait for a vertical retrace */
            while (!(inp(video.isr) & 8));
            /* wait for horizontal or vertical retrace */
            while (inp(video.isr) & 1);
            /* address the preset row scan register */
            outpw(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
            /* Reset the start address */
            set_start_addr(screen.start_addr);
        }
    }
    /* COUNT < 0  => SCROLL SCREEN CHARACTERS DOWNWARDS */
    /* i is the number of scan lines scrolled */
    if (count < 0) {
        if (vpel >= (int) video.char_ht) vpel -= video.char_ht;

        for(i=0;i < -count && screen.start_addr;) {
        /* This loop determines whether to update the start address */
        /* It is iterated once for each video.char_ht. */
            if (vpel < 0) {
                vpel = video.char_ht + vpel;
                if (screen.start_addr > screen.logical_width)
                    screen.start_addr -= screen.logical_width;
                else {
                    screen.start_addr = screen.a_start;
                    vpel=0;
                }
            }
```

(continued on page 110)

Listing Two *(Listing continued, text begins on page 62.)*

```
            /* this loop moves the screen "speed" scan lines each */
            /* time through */
            for(;vpel>=0 && i< -count;vpel-=speed,i+=speed) {
                /* wait for a vertical retrace */
                while (!(inp(video.isr) & 8));
                /* wait for horizontal or vertical retrace */
                while (inp(video.isr) & 1);
                /* address the preset row scan register */
                outpw(video.crtc, (vpel << 8) | PRESET_ROW_SCAN);
                /* Reset the start address */
                set_start_addr(screen.start_addr);
            }
        }
    }
}
/*
SMOOTH_PAN -- This function invokes smooth panning on the EGA/VGA in
text mode.  The function calculates the number of scan lines per row.
The speed variable adjusts the speed in pixels per vertical retrace and
takes values in the range 1-8 for EGA color and 1-9 for monochrome and
the VGA.
*/
int smooth_pan(int count, unsigned speed)
{
    unsigned i;
    /* count greater than zero (move viewport to the right) */
    if (count>0 && !right_edge) for(i=0;i<count;) {
        /* if we have scrolled a full character, reset start address */
        if (hpel >= 8) {
            screen.start_addr++;
            set_start_addr(screen.start_addr);
            if (!left_edge) if (!(screen.start_addr % (screen.logical_
                                                        width)))
                right_edge = TRUE;
            left_edge = FALSE;
            if (!right_edge) {
                if (!(mono || vga.active)) hpel %= 8;     /* reset the */
                                                          /* pel counter */
                else {
                    hpel %= 9;
                    if (hpel == 8) {
                        set_pel_pan(hpel);
                        hpel += speed;
                        hpel %= 9;
                    }
                }
            }
            else {
                hpel = (mono || vga.active) ? 8 : 0;
            }
        }
        for(;hpel < 8 && i < count;i+=speed, hpel+=speed) set_pel_
                                                            pan(hpel);
    }
    /* count less than zero (move viewport to the left) */
    else if (count < 0) {
        if (mono || vga.active) {
            if (left_edge && hpel == 8) return (-1);
            if (hpel > 8) hpel -= 9;
        }
        else {
            if (left_edge && hpel == 0) return (-1);
            if (hpel > 7) hpel = hpel - 8;
        }
        for(i=0; i < (-count); ) {
        /* IF WE HAVE SCROLLED A FULL CHARACTER, RESET START ADDRESS */
            if (hpel<0 && !left_edge) {
                if ((mono || vga.active) && hpel == -1) {
                    hpel = 8;
                    set_pel_pan(hpel);
                    hpel = -1 - speed;
                }
                screen.start_addr--;
                right_edge = FALSE;
                hpel = (mono || vga.active) ? 9 + hpel : 8 + hpel;
                set_start_addr(screen.start_addr);
                if (!(screen.start_addr % screen.logical_width)) {
                    left_edge = TRUE;
                    screen.start_addr--;
                }
            }
            else if (hpel<0 && left_edge) i = (-count);

            for(;hpel >= 0 && i < (-count);i+=speed, hpel-=speed) set_
                                                            pel_pan(hpel);
            if (left_edge && hpel < speed) {
                hpel = (mono || vga.active) ? 8 : 0;
                set_pel_pan(hpel);
                return (-1);
            }
        }
    }
    return (0);
}
/*
SET_LOGICAL_SCREEN_WIDTH defines an EGA/VGA screen width (in bytes) for
smooth panning and sets the the global variable "screen.cols."
screen.cols is the number of screen columns and must be restored when
smooth panning is finished or subsequent screen writes will address the
wrong screen locations.
l_width is the width of the logical screen in bytes. Max 512.
*7
unsigned set_logical_screen_width(unsigned l_width)
{
    l_width = (l_width > 512) ? 512 : l_width;
    /* set screen_columns */
    screen.cols = l_width;
    /* set logical screen width */
    l_width >>= 1;                          /* convert to words */
    outpw(video.crtc, (l_width << 8) | 0x013);

    return (l_width << 1);
}
/*
SET_START_ADDRESS -- Sets the start address of the screen. I.e. the
address that occupies the top left of the screen.
*/
void set_start_addr(unsigned start_addr)
{
    /* address the start address high register */
    outpw(video.crtc, (start_addr & 0XFF00) | START_ADDRESS_HIGH);
    /* address the start address low register */
    outpw(video.crtc, (start_addr << 8) | START_ADDRESS_LOW);
}
/*
SET_PEL_PAN -- Sets the horizontal pel panning register to the value
specified in "hpel".  Called by smooth_pan()
*/
void set_pel_pan(int hpel)
{
    /* first wait for end of vertical retrace */
    while (inp(video.isr) & 8);
    /* wait for next vertical retrace */
    while (!(inp(video.isr) & 8));
    /* address the horizontal pel paning register */
    outp(AC_INDEX, AC_HPP);
    outp(AC_INDEX, hpel);
}
```

End Listings

Listing One *(Text begins on page 81.)*

```
/* ------------ help.h -------------- */

void load_help(char *);
void display_help(void);

extern char *help_window;
#define set_help(s) help_window=s
```

End Listing One

Listing Two

```
/* --------- help.c ----------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include "window.h"
#include "menu.h"
#include "entry.h"
#include "help.h"

#define MAXHELPS 25

static struct helps {
    char hname [9];
    int h, w;
    long hptr;
} hps [MAXHELPS+1];

extern FIELD *fld;
extern MENU *mn;

static int hp = 0;
static FILE *helpfp;
static char hline [80];
char *help_window;

/* ---------- load the HELP! definition file ------------ */
void load_help(char *hn)
{
    extern void (*helpfunc)(void);
    extern int helpkey;
    char *cp;

    helpfunc = display_help;
    helpkey = F1;
    hp = 0;
    if ((helpfp = fopen(hn, "r")) == NULL)
        return;
    if ((fgets(hline, 80, helpfp)) == NULL)
        return;
    while (1)   {
        if (hp == MAXHELPS)
            break;
        if (strncmp(hline, "<end>", 5) == 0)
            break;
        if (*hline != '<')
            continue;
        hps[hp].h = 2;
        hps[hp].w = 23;
        strncpy(hps[hp].hname, hline+1, 8);
        hps[hp].hname[8] = '\0';
        cp = strchr(hps[hp].hname, '>');
        if (cp)
            *cp = '\0';
        hps[hp].hptr = ftell(helpfp);
        if (fgets(hline, 80, helpfp) == NULL)
            strcpy(hline, "<end>");
        while (hline[0] != '<') {
            hps[hp].h++;
            hps[hp].w = max(hps[hp].w, strlen(hline)+2);
        if (fgets(hline, 80, helpfp) == NULL)
            strcpy(hline, "<end>");
        }
        hp++;
    }
}

/* ---------- display the current help window ---------- */
void display_help()
{
    int hx, hy, i, xx, yy, ch;
    extern int helpkey, hsel;
    char *save_help;
    static int inhelp = 0;
    extern struct wn wkw;

    if (inhelp)
        return;
    inhelp++;
    save_help = help_window;
    if (fld != NULL)
        help_window = fld->fhelp;
    else if (mn != NULL)
        help_window = (mn+hsel-1)->mshelp [wkw.wy-1];
```

(continued on page 121)

C PROGRAMMING

Listing Two *(Listing continued, text begins on page 81.)*

```
if (help_window != NULL)    {
    for (ch = 0; ch < hp; ch++)
        if (strcmp(help_window, hps[ch].hname) == 0)
            break;
    if (ch < hp)    {
        xx = wherex();
        yy = wherey();
        hidecursor();
        hx = ((80-hps[ch].w) / 2)+1;
        hy = ((25-hps[ch].h) / 2)+1;
        establish_window(hx, hy,
                hx+hps[ch].w-1, hy+hps[ch].h,
                HELPFG, HELPBG, TRUE);
        fseek(helpfp, hps[ch].hptr, 0);
            for (i = 0; i < hps[ch].h-2; i++)   {
                gotoxy(2,2+i);
                fgets(hline, 80, helpfp);
                cprintf(hline);
            }
            gotoxy(2,2+i);
            cprintf(" [Any key to return]");
            hidecursor();
            getkey();
            delete_window();
            if (mn == NULL || fld != NULL)    {
                textcolor(FIELDFG);
                textbackground(FIELDBG);
                gotoxy(xx, yy);
            }
        }
    }
    help_window = save_help;
    --inhelp;
}
```

End Listing Two

Listing Three

```
<editor>
            TINY WORD PROCESSOR (TWP) COMMANDS
-------Cursor Movement------    ---------Page Movement--------
arrows    = move text cursor    Ctrl-Home = Beginning of File
Ctrl-T    = Top of Window       Ctrl-End  = End of File
Ctrl-B    = Bottom of Window    PgUp      = Previous Page
Ctrl ->   = Next Word           PgDn      = Next Page
Ctrl <-   = Previous Word
Home      = Beginning of Line   ---------Editor Control-------
End       = End of Line         Alt-A  = Auto Paragraph Reform
Shift-Tab = Back tab

--------Block Controls------    ---------Edit Commands--------
F2  = Form Paragraph            Alt-Q or Esc = Done
F5  = Mark Block Beginning      Ins          = Insert Mode
F6  = Mark Block End            Del          = Delete Char
F3  = Move Block                <--          = Rubout
F4  = Copy Block                Ctrl-D       = Delete Word
F8  = Delete Block              Alt-D        = Delete Line
F9  = Unmark Block              F7           = Find
                                Alt-F7       = Find again

<load>
Load a new file into TWP,
replacing the existing file.
<save>
Save the file from the
edit buffer.
<merge>
Merge a file into the
edit buffer at the
line where the cursor
is pointed.
<new>
Clear the edit buffer
and create a new file.
<quit>
Exit from TWP, returning to DOS
<move>
Move the block to the
line where the cursor
points. This is an
insert move.
<copy>
Copy the block to the
line where the cursor
points. This is an
insert copy.
<delete>
Delete the block closing
the space it occupies.
<hide>
Turn off the block
markers.
<formpara>
Form a paragraph.
This makes a paragraph from
a marked block, or, if no
block is marked, to the next
blank or indented line.
<markbeg>
Mark the beginning line
of a block for move, copy,
delete, or paragraph.
<markend>
```

(continued on next page)

Listing Three *(Listing continued, text begins on page 81.)*

```
Mark the ending line
of a block for move, copy,
delete, or paragraph.
<find>
Find a specified string
in the edit buffer.
Move the cursor to the
location where the string
was found.
<findagn>
Find the next occurrence
of the string most recently
specified.
<auto>
Turn on/off the automatic
paragraph forming feature.
<insert>
Turn on/off the
character insert mode.
(insert/overstrike toggle)
<filename>
Enter the path and file name.
The path is optional but must
be fully qualified if entered.
<findstr>
Enter the string to be searched
by the Find command.
<end>
```

End Listing Three

Listing Four

```
/* --------- editshel.c ------------ */
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <mem.h>
#include <alloc.h>
#include <ctype.h>
#include "window.h"
#include "menu.h"
#include "entry.h"
#include "editor.h"
#include "help.h"

int MAXLINES;           /* maximum number of editor lines */
#define EDITWIDTH  78   /* length of an editor line       */
#define BUFLEN (EDITWIDTH*MAXLINES)

/* --------- alt keys returned by getkey() --------- */
#define ALT_A 158
#define ALT_L 166
#define ALT_M 178
#define ALT_N 177

/* -------- configured advanced editor commands --------- */
#define FIND            F7
#define FIND_AGAIN      ALT_F7
#define LOAD_FILE       ALT_L
#define SAVE_FILE       ALT_S
#define MERGE_FILE      ALT_M
#define NEW_FILE        ALT_N
#define EDITOR_MENU     F10
#define REFORM          ALT_A

/* ---------- editor menu tables --------- */
static char *fselcs[] = {
    "Load   [Alt-L]",
    "Save   [Alt-S]",
    "Merge  [Alt-M]",
    "New    [Alt-N]",
    "Quit   [Alt-Q]",
    NULL
};

static char *filehelp[] = {
    "load",
    "save",
    "merge",
    "new",
    "quit"
};

static char *eselcs[] = {
    "Move          [F3]",
    "Copy          [F4]",
    "Delete        [F8]",
    "Hide          [F9]",
    "Paragraph     [F2]",
    "Mark Beg      [F5]",
    "Mark End      [F6]",
    "Find          [F7]",
    "Find Again    [Alt-F7]",
    NULL
};

static char *edithelp[] = {
    "move",
    "copy",
    "delete",
    "hide",
    "formpara",
    "markbeg",
    "markend",
    "find",
    "findagn",
};

static char *oselcs[] = {
    "Auto Paragraph Reformat   [Alt-A]",
    "Insert                    [Ins]",
    NULL
};

static char *opthelp[] = {
    "auto",
    "insert"
};

void fileedit(char *);
static int  edit(int,int);
static void editmenu(int);
static int  get_filename(char *, int);
static int  write_file(void);
static int  load_file(int,int);
static int  save_file(int,int);
static int  merge_file(int,int);
static int  new_file(int,int);
static void editkeys(int);
static void statusline(void);
static int  findstring(void);
static char *find(char *, unsigned);
static int  read_file(char *, char *, int, int, int);
static int  bufferok(char *);
static void notice(char *);
void (*edit_extend)(void);

static char fkeys[] =   {LOAD_FILE,SAVE_FILE,
                         MERGE_FILE,NEW_FILE,QUIT};
static char forced[] =  {MOVE_BLOCK,COPY_BLOCK,
                         DELETE_BLOCK,HIDE_BLOCK,
                         PARAGRAPH,BEGIN_BLOCK,
                         END_BLOCK,FIND,FIND_AGAIN};
static char options[] = {REFORM,INS};

static int (*ffuncs[])() =
    {load_file,save_file,merge_file,new_file,edit};
static int (*efuncs[])() =
    {edit,edit,edit,edit,edit,edit,edit,edit,edit};
static int (*ofuncs[])() =
    {edit,edit,edit};

MENU emn [] = {
    {"File",    NULL, fselcs, filehelp, fkeys,   ffuncs, 0},
    {"Edit",    NULL, eselcs, edithelp, forced,  efuncs, 0},
    {"Options", NULL, oselcs, opthelp,  options, ofuncs, 0},
    {NULL}
};

/* ------ filename data entry template and buffer ------- */
static char filename[65];
static char savefn [65];
static char filemask[65];

FIELD fn_template[] = {
    {2,14,1,filename,filemask,"filename"},
    {0}
};

/* ------- text find data entry template and buffer ------ */
static char findstr[71];
static char findmask[71];

FIELD find_template[] = {
    { 2,8,1,findstr,findmask,"findstr"},
    {0}
};

extern int forcechar;
extern struct edit_env ev;

static void editkeys(int c)
{
    switch(c)   {
        case REFORM:
            ev.reforming ^= TRUE;
            break;
        case NEW_FILE:
            new_file(1,1);
            break;
        case LOAD_FILE:
            load_file(1,1);
            break;
        case SAVE_FILE:
            save_file(1,1);
            break;
        case MERGE_FILE:
            merge_file(1,1);
            break;
        case FIND:
            if (!findstring())
                break;
        case FIND_AGAIN:
            ev.nowptr++;
            ev.nowptr = find(ev.nowptr, ev.lstptr-ev.nowptr);
            if (ev.nowptr != NULL)  {
                ev.curr_x = (ev.nowptr-ev.topptr) % ev.wwd;
                if (ev.nowptr >= ev.bfptr+ev.wwd*ev.wdo->ht)
                    ev.bfptr = ev.nowptr - ev.curr_x;
                ev.curr_y = (ev.nowptr - ev.bfptr) / ev.wwd;
            }
            else
                error_message("Not found ...");
            break;
        case ALT_F:
```

```
        editmenu(1);
        break;
    case ALT_E:
        editmenu(2);
        break;
    case ALT_O:
        editmenu(3);
        break;
    case EDITOR_MENU:
        editmenu(0);
        break;
    default:
        if (edit_extend)
            (*edit_extend)();
        else
            putch(BELL);
                break;
        }
    }

    static void editmenu(n)
    {
        menu_select(emn, n);
    }

    static int edit(hs,vs)
    {
        forcechar = emn[hs-1].mskeys[vs-1] & 255;
        return TRUE;
    }
    /* ---------- edit a file -------------- */
    void fileedit(char *file)
    {
        char *bf, *mb;
        extern void (*editfunc)();
        extern void (*status_line)();

        setmem(filename, 64, ' ');
        setmem(filemask, 64, ' ');
        setmem(findmask, 70, '_');
        setmem(findstr, 70, ' ');
        establish_window(1,2,80,24,TEXTFG,TEXTBG,TRUE);
        editfunc = editkeys;
        status_line = statusline;
        mb = display_menubar(emn);
        statusline();
        if ((bf = malloc(BUFLEN)) != NULL)  {
            setmem(bf, BUFLEN, ' ');
            strcpy(filename, file);
            filename[strlen(filename)] = ' ';
            if (*file)
                read_file(" Loading ... ",bf,0,FALSE,FALSE);
            while (TRUE)    {
                text_editor(bf, MAXLINES, EDITWIDTH);
                if (bufferok("quit"))
                    break;
            }
            free(bf);
        }
        restore_menubar(mb);
        delete_window();
    }
    /* ---------- load a file --------------- */
    static int load_file(hs,vs)
    {
        if (bufferok("reload"))
            strcpy(savefn, filename);
            if (get_filename(" Load what file? ", TRUE) != ESC) {
                setmem(ev.topptr, BUFLEN, ' ');
                read_file(" Loading ... ",ev.topptr,0,FALSE,TRUE);
                forcechar = BEGIN_BUFFER;
                ev.text_changed = FALSE;
            }
            else
                strcpy(filename, savefn);
        return TRUE;
    }
    /* ---------- merge a file into the edit buffer -------- */
    static int merge_file(hs,vs)
    {
        strcpy(savefn, filename);
        if (get_filename(" Merge what file? ", TRUE) != ESC)    {
            if (read_file(" Merging ... ",
                    curr(0, ev.curr_y),
                    lineno(ev.curr_y), TRUE, TRUE)) {
                forcechar = REPAINT;
                ev.text_changed = TRUE;
            }
        }
        strcpy(filename, savefn);
        return TRUE;
    }
    /* --------- save the file -------------- */
    static int save_file(hs,vs)
    {
        if (get_filename(" Save as what file? ", FALSE) != ESC)
            if (write_file())
                ev.text_changed = FALSE;
        return TRUE;
    }
    /* ---------- start a new file ------------ */
    static int new_file(hs,vs)
    {
        if (bufferok("erase"))
            if (get_filename(" Build as what file? ",TRUE)!=ESC){
                setmem(ev.topptr, BUFLEN, ' ');
```

(continued on next page)

Listing Four *(Listing continued, text begins on page 81.)*

```
                forcechar = BEGIN_BUFFER;
                ev.text_changed = FALSE;
            }
        return TRUE;
    }
    /* -------- read a file name ------------ */
    static int get_filename(char *ttl, int clear)
    {
        int rtn;

        establish_window(1,23,80,25,ENTRYFG,ENTRYBG,TRUE);
        window_title(ttl);
        gotoxy(3,2);
        cputs("File name:");
        rtn = data_entry(fn_template, clear, 1);
        delete_window();
        return rtn;
    }

    /* --------- write a file ------------ */
    static int write_file()
    {
        FILE *fp;
        int ln, i, ln1;
        char *cp, buf[EDITWIDTH+1];

        if ((fp = fopen(filename, "w")) == NULL)    {
            error_message(" Can't write that file! ");
            return FALSE;
        }
        notice(" Writing file ... ");
        /* ----- find the last significant line ----- */
        for (ln = MAXLINES-1; ln > -1; --ln)    {
            cp = ev.topptr + ln * EDITWIDTH;
            for (i = 0; i < EDITWIDTH; i++)
                if (*(cp + i) != ' ')
                    break;
            if (i < EDITWIDTH)
                break;
        }
        for (ln1 = 0; ln1 <= ln; ln1++) {
            movmem(ev.topptr + ln1 * EDITWIDTH, buf, EDITWIDTH);
            i = EDITWIDTH-1;
            cp = buf;
            while (i >= 0 && *(cp + i) == ' ')
                --i;
            if (i == -1 || *(cp + i) != ' ')
                i++;
            *(cp + i) = '\n';
            *(cp + i + 1) = '\0';
            fputs(cp, fp);
        }
        fclose(fp);
        delete_window();
        return TRUE;
    }

    /* -------------- read (load or merge) a file ----------- */
    static int
    read_file(char *nt,char *ln,int lines,int merging,int needed)
    {
        FILE *fp;
        char ibf[120];
        char *cp;
        int x;

        if ((fp = fopen(filename, "r")) != NULL)    {
            notice(nt);
            while (fgets(ibf, 120, fp) && lines < MAXLINES) {
                lines++;
                if (merging)    {
                    movmem(ln,ln+EDITWIDTH,
                        BUFLEN-lines*EDITWIDTH);
                    setmem(ln,EDITWIDTH,' ');
                }
                cp = ibf, x = 0;
                while (*cp && *cp != '\n')  {
                    if (*cp == '\t')
                        x += TAB-(x%TAB);
                    else
                        *(ln+x++) = *cp;
                    cp++;
                }
                ln += EDITWIDTH;
            }
            fclose(fp);
            delete_window();
            return TRUE;
        }
        else if (needed)
            error_message("No such file can be found");
        return FALSE;
    }

    /* ---------- display a status line ---------- */
    static void statusline()
    {
        char stat[81], *st;
        int cl[81], *cp;
        static char msk[] =
    "Line:%3d   Column:%2d   %-9.9s   %-21.21s   F1:Help    \
    F10:Menu  ";
        unsigned y = 1;
        unsigned x = 1;
        unsigned attr = ((MENUFG | (MENUBG << 4)) << 8);
    if (ev.wwd) {
        y = (unsigned) (ev.nowptr-ev.topptr) / ev.wwd + 1;
        x = (unsigned) (ev.nowptr-ev.topptr) % ev.wwd + 1;
    }
    sprintf(stat,msk,y,x,
        (ev.edinsert ? "Insert" : "Overwrite"),
```

```
        (ev.reforming ? "Auto Paragraph Reform" : " "));
    for (st = stat, cp = cl; *st; st++)
        *cp++ = (*st & 255) | attr;
    _vram(_vptr(1,25),cl,80);
    set_help("editor");
}

/* -------- get a string to find --------- */
static int findstring()
{
    char *cp = findstr+60;
    int ans;

    establish_window(1,23,80,25,ENTRYFG,ENTRYBG,TRUE);
    gotoxy(2,2);
    cputs("Find?");
    ans = data_entry(find_template, TRUE, 1);
    delete_window();
    if (ans == ESC)
        return FALSE;
    while (*--cp == ' ')
        ;
    if (*cp)
        *(cp+1) = '\0';
    return TRUE;
}

/* -------- find a string in the buffer -------------- */
static char *find(char *bf, unsigned len)
{
    char *cp;

    for (cp = bf; cp < bf+len-strlen(findstr); cp++)
        if (strncmp(cp, findstr, strlen(findstr)) == 0)
            return cp;
    return NULL;
}

/* ---------- test for buffer changed ----------- */
static int bufferok(char *s)
{
    int c = 'Y';
    if (ev.text_changed)    {
        establish_window(23,11,56,13,ERRORFG,ERRORBG,TRUE);
        gotoxy(2,2);
        cprintf("Text has changed, %s? (y/n)", s);
        hidecursor();
        do
            putch(BELL), c = getkey();
        while (toupper(c) != 'Y' && toupper(c) != 'N');
        delete_window();
    }
    return toupper(c) == 'Y';
}
```

```
/* -------- small message ------------ */
static void notice(char *s)
{
    int lf = (80-strlen(s))/2-1;
    int rt = lf+strlen(s)+2;
    establish_window(lf,11,rt,13,HELPFG,HELPBG,TRUE);
    gotoxy(2,2);
    cputs(s);
}
```

End Listing Four

Listing Five

```
/* ---------- twrp.c ------------ */
#include <conio.h>
#include "window.h"
#include "editor.h"
#include "help.h"

void main(int, char **);
void fileedit(char *);

void main(int argc, char **argv)
{
    extern int inserting, MAXLINES;

    MAXLINES = 800;
    load_help("twrp.hlp");
    clear_screen();
    fileedit(argc > 1 ? argv[1] : "");
    clear_screen();
    inserting = FALSE;
    insert_line();
}
```

End Listing Five

Listing Six

```
twrp (window.h, editor.h, help.h)
editshel (editor.h, menu.h, entry.h, help.h, window.h)
editor (editor.h, window.h)
entry (entry.h, window.h)
menu (menu.h, window.h)
help (help.h, window.h)
window (window.h)
```

End Listings

STRUCTURED PROGRAMMING

Listing One *(Text begins on page 86.)*

```
 1| Program number;
 2|
 3|   { Puts line numbers at start of each line, stores in file by same
 4|     name except extension is .NUM }
 5|
 6| USES crt;
 7|
 8| VAR   Filename, Newname : STRING [80];
 9|       I, O             : TEXT;
10|       Line             : STRING [135];
11|       Nbr, len, p      : WORD;
12|       Num              : STRING [4];
13|
14| BEGIN
15|   Nbr := 0;
16|   Newname := '';
17|   IF ParamCount < 1 THEN BEGIN
18|     Writeln ('USAGE: NUMBER <Filename.ext>');
19|     EXIT;
20|   END;
21|   Filename := ParamStr (1);
22|
23|   Len := pos ('.', Filename);
24|   IF len = 0 THEN
25|     Newname := Filename + '.NUM'
26|   ELSE BEGIN
27|     FOR p := 1 TO len DO
28|       Newname := Newname + Filename [p];
29|     Newname := Newname + 'NUM';
30|   END;
31|   Assign (I, Filename);
32|   {$I-}
33|   Reset (I);
34|   {$I+}
35|   IF IOResult <> 0 THEN BEGIN
36|     Writeln ('Unable to open ', Filename);
37|     EXIT;
38|   END;
39|   Assign (O, Newname);
40|   Rewrite (O);
41|   Writeln;
42|
43|   WHILE NOT eof (I) DO BEGIN
44|     Readln (I, Line);
45|     INC (Nbr);
46|     GotoXY (1, WhereY-1); Writeln (Nbr);
47|     Str (Nbr:4, Num);
48|     Line := Num + '| ' + Line;
49|     Writeln (O, Line);
50|   END;
51|   Close (O);
52|   Close (I);
53|   GotoXY (1, WhereY-1); Writeln (nbr, ' lines in file');
54|   Writeln ('Output is in ', Newname);
55| END.
56|
57|
```

End Listing One

Listing Two

```
Case-sensitive symbolic cross-reference for number.pas

0 (3), 15, 24, 35
1 (7), 17, 21, 27, 46, 46, 53, 53
135 (1), 10
4 (2), 12, 47
80 (1), 8
Assign (2), 31, 39
Close (2), 51, 52
crt (1), 6
eof (1), 43
EXIT (2), 19, 37
Filename (7), 8, 21, 23, 25, 28, 31, 36
GotoXY (2), 46, 53
I (6), 9, 31, 33, 43, 44, 52
IOResult (1), 35
Len (1), 23
len (3), 11, 24, 27
Line (5), 10, 44, 48, 48, 49
Nbr (5), 11, 15, 45, 46, 47
nbr (1), 53
Newname (9), 8, 16, 25, 28, 28, 29, 29, 39, 54
NOT (1), 43
Num (3), 12, 47, 48
number (1), 1
O (5), 9, 39, 40, 49, 51
p (3), 11, 27, 28
ParamCount (1), 17
ParamStr (1), 21
pos (1), 23
Program (1), 1
Readln (1), 44
Reset (1), 33
Rewrite (1), 40
Str (1), 47
STRING (3), 8, 10, 12
TEXT (1), 9
USES (1), 6
WhereY (2), 46, 53
WORD (1), 11
```

```
Writeln (7), 18, 36, 41, 46, 49, 53, 54

-- 40 symbols reported
```

End Listing Two

Listing Three

```
 1| PROGRAM Xref;
 2|
 3| { Builds and lists a Pascal/Modula-2 symbol cross-reference report   }
 4| { Uses binary trees and doubly-linked lists to effect B-Tree        }
 5| { Command line is XREF <filename.ext> [/C|/N]                       }
 6| { /C makes xref case-sensitive                                      }
 7| { /N makes it non-case sensitive (default)                          }
 8| { Turbo Pascal 5.0 (4.0 will work, too)                             }
 9| { K. Porter, DDJ, December '88 Structured Programming Column        }
10|
11| USES crt, printer;
12|
13| TYPE  SymString = STRING [39];
14|       CharSet = SET OF CHAR;
15|       LineString = STRING [135];
16|       XLinePtr = ^XLineNode;       { Pointer to xref line number node }
17|       XLineNode = RECORD           { Xref line number structure (SLL) }
18|         Line : WORD;
19|         Next : XLinePtr;
20|       END;
21|
22|       SymTreePtr = ^SymTreeNode;       { Pointer to symbol tree node }
23|       SymTreeNode = RECORD                { Binary tree symbol node }
24|         Symbol       : SymString;
25|         UCsymbol     : SymString;
26|         Count        : WORD;
27|         XList        : XLinePtr;
28|         LLink, RLink : SymTreePtr;
29|       END;
30|
31| CONST Quote =  #39;
32|       DQuote = #34;
33|       Eject =  #12;
34|       SymChars : CharSet = ['0'..'9','A'..'Z','a'..'z','.','_','^'];
35|       PComment : CharSet = ['{', '}', '(', '*', ')', Quote, DQuote];
36|       Heading  = ' symbolic cross-reference for ';
37|
38| VAR   Filepath       : STRING [80];
39|       Case_Sensitive : BOOLEAN;
40|       F              : TEXT;
41|       Head, Alpha    : SymTreePtr;
42|       CommentLevel   : WORD;
43|       Line           : LineString;
44|       LineNumber     : WORD;
45|       NSymbols       : WORD;
46| { ------------------------------------------------------------------ }
47|
48| PROCEDURE FindEndOfComment (VAR line : LineString;
49|                             VAR i    : WORD;
50|                                 eoc  : CHAR);
51|   { Scan until end of current comment is found }
52|
53| VAR   ch        : CHAR;
54|       Searching : BOOLEAN;
55|
56| BEGIN
57|   Searching := TRUE;
58|   WHILE Searching DO BEGIN
59|     WHILE i <= Length (Line) DO BEGIN
60|       ch := Line [i];
61|       INC (i);
62|       IF ch = eoc THEN
63|         CASE eoc OF
64|           '}':    Searching := FALSE;
65|           '*':    IF line [i] = ')' THEN BEGIN
66|                     Searching := FALSE;
67|                     INC (i);
68|                   END;
69|           Quote:  Searching := FALSE;
70|           DQuote: Searching := FALSE;
71|         END;
72|
73|       IF Searching = FALSE THEN BEGIN
74|         DEC (CommentLevel);
75|         EXIT;
76|       END;
77|     END;
78|
79|     { If we get here, the comment continues on the next line }
80|     Readln (F, Line);
81|     i := 1;
82|     INC (LineNumber);
83|   END;
84| END;
85| { -------------------------- }
86|
87| FUNCTION UpShift (VAR Symbol : SymString) : SymString;
88|   { Return upshifted version of passed string }
89|
90| VAR   p : INTEGER;
91|       s : SymString;
92|
93| BEGIN
94|   s := '';
95|   FOR p := 1 TO Length (Symbol) DO
96|     s := s + UpCase (Symbol [p]);
97|   UpShift := s;
98| END;
99| { -------------------------- }
100|
101| FUNCTION NewNode (VAR Symbol : SymString) : SymTreePtr;
102|   { Allocate and set up new symbol node, return pointer }
103|
104| VAR   node : SymTreePtr;
105|
106| BEGIN
107|   NEW (node);
108|   Node^.Symbol   := Symbol;
109|   Node^.UCSymbol := UpShift (Symbol);
110|   Node^.Count    := 1;
111|   Node^.XList    := NIL;
112|   Node^.RLink    := NIL;
113|   Node^.LLink    := NIL;
114|   Node^.RLink    := NIL;
115|   NewNode := node;
116| END;
117| { -------------------------- }
118|
119| FUNCTION Token (VAR line : LineString;
120|                 VAR i : WORD) : SymString;
121|   { Return next symbol or keyword from line }
122|   { Index to next char returned as a side-effect }
123|   { Also checks for comments }
124|
125| VAR   sym          : SymString;
126|       ch, ScanChar : CHAR;
127|       nch          : WORD;
128|
129| BEGIN
130|   { Scan for first valid alphanumeric or for comment }
131|   ScanChar := #0;
132|   WHILE (NOT (Line [i] IN SymChars)) AND (i <= Length (line)) DO BEGIN
133|     ch := line [i];
134|     INC (i);
135|     IF ch IN PComment THEN BEGIN
136|       CASE ch OF
137|         Quote: BEGIN
138|                  INC (CommentLevel);
139|                  ScanChar := Quote;
140|                END;
141|         '{':   BEGIN
142|                  INC (CommentLevel);
143|                  ScanChar := '}';
144|                END;
145|         '}':   IF CommentLevel > 0 THEN
146|                  DEC (CommentLevel);
147|         '(':   IF line [i] = '*' THEN BEGIN
148|                  INC (CommentLevel);
149|                  ScanChar := '*';
150|                  INC (i);
151|                END;
152|         '*':   IF line [i] = ')' THEN
153|                  IF CommentLevel > 0 THEN BEGIN
154|                    DEC (CommentLevel);
155|                    INC (i);
156|                  END;
157|       END;
158|       IF CommentLevel > 0 THEN
159|         FindEndOfComment (line, i, ScanChar);
160|     END;
161|   END;
162|
163|   { Pull out the symbol }
164|   sym := '';
165|   nch := 1;
166|   IF i < Length (Line) THEN
167|     REPEAT
168|       ch := Line [i];
169|       IF ch IN SymChars THEN BEGIN
170|         IF (ch = '^') AND (nch = 1) THEN
171|           { Skip leading pointer char }
172|         ELSE BEGIN
173|           sym := sym + ch;
174|           INC (nch);
175|         END;
176|         INC (i);
177|       END;
178|     UNTIL (NOT (ch IN SymChars)) OR (i > Length (Line));
179|   IF NOT Case_Sensitive THEN
180|     Token := UpShift (sym)
181|   ELSE
182|     Token := sym;
183| END;
184| { -------------------------- }
185|
186| FUNCTION BNode (VAR sym : SymString) : SymTreePtr;
187|   { Find sym's node in binary tree, or add it if it doesn't exist }
188|
189| VAR   Node, Parent : SymTreePtr;
190|
191| BEGIN
192|   Node := Head;
193|   WHILE ((Node^.Symbol <> sym) AND (Node <> NIL)) DO BEGIN
194|     Parent := Node;
195|     IF sym < Node^.Symbol THEN
196|       Node := Node^.LLink
197|     ELSE
198|       Node := Node^.RLink
199|   END;
200|   IF Node <> NIL THEN                    { Node exists for this symbol }
201|     INC (Node^.Count)
202|   ELSE BEGIN                          { Else add new node to binary tree }
203|     Node := NewNode (sym);
204|     IF sym < Parent^.Symbol THEN           { Update parent's pointer }
205|       Parent^.LLink := Node
206|     ELSE
207|       Parent^.RLink := Node
208|   END;
```

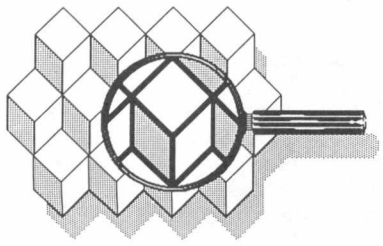
(continued on page 130)

Listing Three *(Listing continued, text begins on page 86.)*

```
209|   BNode := Node;
210| END;
211| { ---------------------------- }
212|
213| PROCEDURE Append (Target : SymTreePtr; LineNbr : WORD);
214|   { Add line cross-ref to target's dependent list }
215|
216| VAR    XR, Parent : XLinePtr;
217|
218| BEGIN
219|   IF Target^.XList = NIL THEN BEGIN      { First occurrence of symbol }
220|       NEW (XR);
221|       XR^.Line := LineNbr;
222|       XR^.Next := NIL;
223|       Target^.XList := XR;
224|     END
225|   ELSE BEGIN                           { Append to end of existing list }
226|       XR := Target^.Xlist;
227|       REPEAT
228|         Parent := XR;
229|         XR := XR^.Next
230|       UNTIL XR = NIL;                                { Find list's tail }
231|       NEW (XR);                                        { Append there }
232|       XR^.Line := LineNbr;
233|       XR^.Next := NIL;
234|       Parent^.Next := XR;
235|   END;
236| END;
237| { ---------------------------- }
238|
239| PROCEDURE AddToTree (VAR Symbol : SymString; LineNbr : WORD);
240|   { Place symbol into binary tree, add line xref to dependent list }
241|
242| VAR   Target : SymTreePtr;
243|
244| BEGIN
245|   IF Head = NIL THEN BEGIN          { The tree is empty, so start it }
246|       Head := NewNode (Symbol);                { Build first binary node }
247|       Append (Head, LineNbr);                    { Build first XREF node }
248|     END
249|   ELSE BEGIN
250|       Target := BNode (Symbol);
251|       Append (Target, LineNbr);
252|   END;
253| END;
254| { ---------------------------- }
255|
256| PROCEDURE Process (VAR Line : LineString);
257|   { Controls parsing and construction of BTree }
258|
259| VAR   Symbol  : SymString;
260|       p, oldp : WORD;
261|
262| BEGIN
263|   p := 1;
264|   IF Length (Line) > 0 THEN
265|     WHILE p <= Length (Line) DO BEGIN
266|       oldp := p;
267|       Symbol := Token (line, p);                          { Get symbol }
268|       IF Symbol = 'BEGIN' THEN Symbol := ''      { Weed out nuisances }
269|       ELSE IF Symbol = 'END'     THEN Symbol := ''
270|       ELSE IF Symbol = 'IF'      THEN Symbol := ''
271|       ELSE IF Symbol = 'THEN'    THEN Symbol := ''
272|       ELSE IF Symbol = 'ELSE'    THEN Symbol := ''
273|       ELSE IF Symbol = 'DO'      THEN Symbol := ''
274|       ELSE IF Symbol = 'WHILE'   THEN Symbol := ''
275|       ELSE IF Symbol = 'FOR'     THEN Symbol := ''
276|       ELSE IF Symbol = 'TO'      THEN Symbol := ''
277|       ELSE IF Symbol = 'VAR'     THEN Symbol := ''
278|       ELSE IF Symbol = 'INC'     THEN Symbol := ''
279|       ELSE IF Symbol = 'DEC'     THEN Symbol := ''
280|       ELSE IF Symbol = 'OF'      THEN Symbol := ''
281|       ELSE IF Symbol = 'PROGRAM' THEN Symbol := ''
282|       ELSE IF Symbol = 'END.'    THEN Symbol := '';
283|       IF Length (Symbol) > 0 THEN
284|         AddToTree (Symbol, LineNumber);          { Place info in BTree }
285|       IF p = oldp THEN INC (p);                { Prevents endless loop }
286|     END;
287| END;
288| { ---------------------------- }
289|
290| PROCEDURE Report (Node : SymTreePtr);
291|   { Print symbol cross-reference listing }
292|   { In-order (recursive) traversal of binary tree, printing the info
293|       and dependent list for each node }
294|
295| VAR   Col, Width : WORD;
296|       Lnode      : XLinePtr;
297|
298|   PROCEDURE NewLine;
299|     { Control pagination }
300|   BEGIN
301|     Writeln (LST);
302|     Col := 0;
303|     INC (LineNumber);
304|     IF LineNumber > 58 THEN BEGIN
305|       Write (LST, Eject);
306|       Writeln (LST, 'Continuing cross-reference for ', Filepath);
307|       Writeln (LST);
308|       LineNumber := 2;
309|     END;
310|   END;                                          { End nested procedure }
311|
312| BEGIN
313|   IF node <> NIL THEN BEGIN
314|     Report (Node^.LLink);                        { Follow left-hand path }
315|
316|     { Print info from node }
317|     Col := 0;
318|     Write (LST, Node^.Symbol, ' (', Node^.Count, ')');
319|     Col := Col + Length (Node^.Symbol) + 6;
320|
321|     { Print line number references }
322|     Lnode := Node^.XList;
323|     While Lnode <> NIL DO BEGIN
324|       IF Col > 0 THEN
325|         Write (LST, ', ', Lnode^.Line)
326|       ELSE
327|         Write (LST, '  ', Lnode^.Line);
328|       IF Lnode^.Line < 10 THEN Width := 1
329|         ELSE IF Lnode^.Line > 99 THEN Width := 3
330|           ELSE Width := 2;
331|       Col := Col + Width + 2;
332|       IF (Col > 70) AND (Lnode^.Next <> NIL) THEN NewLine;
333|       Lnode := Lnode^.Next;
334|     END;
335|     NewLine;
336|
337|     Report (Node^.RLink);                 { Then follow right-hand path }
338|   END;
339| END;
340| { ---------------------------- }
341|
342| PROCEDURE Alphabetize (sym : SymTreePtr);
343|   { Rearrange tree when case-sensitive so that upper- and lower-case
344|     identifiers occur in alphabetic order regardless of case }
345|   { RECURSIVE: Traverses symbol table in-order, builds alpha list }
346|
347|   PROCEDURE Resort (sym : SymTreePtr);
348|     { NESTED: Place new node in tree headed by Alpha pointer }
349|
350|   VAR   Node, Parent : SymTreePtr;
351|         UCsymbol     : SymString;
352|
353|   BEGIN
354|     IF Alpha = NIL THEN BEGIN         { Make first node in sorted tree }
355|         Alpha := NewNode (sym^.symbol);
356|         Alpha^.count     := sym^.count;
357|         Alpha^.XList     := sym^.XList;
358|       END
359|     ELSE BEGIN                                 { Add new node in order }
360|         UCsymbol := UpShift (sym^.symbol);
361|         Node := Alpha;
362|         WHILE node <> NIL DO BEGIN              { Find insertion point }
363|           Parent := node;
364|           IF UCsymbol < Node^.UCsymbol THEN     { based on U/C symbol }
365|             Node := Parent^.LLink
366|           ELSE
367|             Node := Parent^.RLink;
368|         END;
369|         Node := NewNode (sym^.symbol);                       { Add node }
370|         Node^.Count     := sym^.count;
371|         Node^.XList     := sym^.XList;
372|         IF UCsymbol < Parent^.UCsymbol THEN
373|           Parent^.LLink := node
374|         ELSE
375|           Parent^.RLink := node;
376|       END;
377|   END;
378|
379| BEGIN  { Body of Alphabetize }
380|   IF sym <> NIL THEN BEGIN
381|     Alphabetize (sym^.LLink);                        { Do nodes to left }
382|     Resort (sym);                                  { Realphabetize this node }
383|     Alphabetize (sym^.RLink);                       { Now do nodes to right }
384|     Dispose (sym);                                 { All thru with this node }
385|   END;
386| END;
387| { ---------------------------- }
388|
389| PROCEDURE Count (Node : SymTreePtr);
390|   { Count nodes in tree }
391| BEGIN
392|   IF node <> NIL THEN BEGIN
393|     Count (Node^.LLink);
394|     INC (NSymbols);
395|     Count (Node^.RLink);
396|   END
397| END;
398| { ---------------------------- }
399|
400| BEGIN
401|   { Initialize }
402|   Head := NIL;
403|   Alpha := NIL;
404|   CommentLevel := 0;
405|   LineNumber := 0;
406|   NSymbols := 0;
407|
408|   { Process command line }
409|   IF ParamCount < 1 THEN BEGIN
410|     Writeln ('USAGE: XREF <Filename.[ext]> [/C|/N]');
411|     EXIT;
412|   END;
413|   Filepath := ParamStr (1);
414|   IF pos ('.', Filepath) = 0 THEN
415|     Filepath := Filepath + '.PAS';          { Default is Pascal file }
416|   Case_Sensitive := FALSE;                { Set default case sensitivity }
417|   IF ParamCount = 2 THEN                    { or alter per command line }
418|     IF (ParamStr (2) = '/c') OR (ParamStr (2) = '/C') THEN
419|       Case_Sensitive := TRUE;
420|
421|   { Open the file }
```

(continued on page 138)

OF INTEREST



Programming Utilities
Microsoft has shipped a new release of the Software Development Kit for Microsoft OS/2. The release contains an updated version of MS OS/2, Version 1.1, including Presentation Manager, updated development tools, and new sample programs. Development tools that have been updated include the Font Editor, Dialog Editor, and Icon Editor.

The latest documentation is provided through an electronic QuickHelp program that can be run as an OS/2 window or called from the command line. The QuickHelp databases contain the complete programmer's reference, as well as additional information describing the development tools in the MS OS/2, Version 1.1, Toolkit.

The MS OS/2 Software Development Kit is available for $3,000. The SDK includes OS/2 software, documentation, utilities, sample code, online electronic support, training videotapes, and updates. For existing SDK licensees, additional kits (with online support and videotapes) are available for $1,500. Reader Service No. 24.
Microsoft Corp.
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717
206-882-8080

Texas Instruments has introduced a set of tools to assist software developers producing applications based on the TI TMS34010 Graphics System Processor. The new 34010 Software Developer's Kit includes an assembler package, a C compiler package, a math/graphics library, a font library, PC utilities, and documentation.

The assembler package translates assembly source modules into 34010-executable COFF (common object file format) section-oriented object code. Its macro-assembler/linker supports packed bit fields, cache alignment, symbol cross-referencing, and output control. It also includes a source/object archiver and a utility for ROM, PROM, or EPROM programming.

The math/graphics library provides over 100 software routines, including functions for graphics drawing, system initialization, 3-D matrix transformation, text control, pixel array, viewpoint management, color palette control, transcendental math operations, and a "paint" application example. The font library includes over 100 sets of bit-mapped images representing characters for 19 different font styles, and provides families of characters suitable for body text, headings, math symbology, figure labels, or any other applications of text in graphics systems.

Available as an option to the kit is the 34010 Software Development Board (SDB), a tool for evaluating and developing software to operate on 34010-based systems. The board operates in IBM PC, XT, AT, and most compatibles. The SDB directly drives most digital and analog R-G-B raster-scan monitors at 640 × 480 × 4 resolution. The board also comes with an interactive PC-user interface that offers a command set for software debugging.

The 34010 Software Developer's Kit is available at a suggested retail price of $1,495. Reader Service No. 25.
Texas Instruments
Semiconductor Group SC-861
P.O. Box 809066
Dallas, TX 75380-9066
800-232-3200, ext. 700

Turbo Apprentice 5.0, released by **Cypress Systems**, allows users to catalog and reference their own units and procedures. Because this product works directly from program and library files, users do not have to create a separate help database.

Priced at $25, Turbo Apprentice requires Turbo Pascal 4.0 and MS-DOS 2.0 or later. It runs on IBM PCs or compatibles. Reader Service No. 20.
Cypress Systems Inc.
11693 San Vicente Blvd., Ste. 175
Los Angeles, CA 90049
213-207-3938

Lulay Software is offering a new product called Lulay Backdoor, a hexadecimal editor designed for heavy use. Features include instant help screens, a built-in ASCII table, and the ability to set LaserJet printer control files, allocate new files, send hex strings to output devices or files, find system or hidden files, and change file attributes.

Costing $89.95, Lulay Backdoor works on local area networks as well as single-user machines. Reader Service No. 21.
Lulay Software
1944 SE Clatsop
Portland, OR 97202
800-336-1166

V.I. Corp. has announced an editor construction kit, which is an expansion to their family of Data Views graphics products. The new product will allow programmers to build an application-specific drawing editor to edit object specific to an application.

The editor construction kit will be available in the first quarter of 1989 and is expected to cost $2,000–4,000. Reader Service No. 22.
V.I. Corp.
Amherst Research Park
Amherst, MA 01002
413-253-3482

NuTools: Numerical Methods in C, a numerical methods library for THINK's LightspeedC, has been released by **Metaphor**. The new product provides developers with an assortment of over 350 numerical routines and includes over 50 input/output routines.

Both low- and medium-level numerical operations are provided for real, complex, polynomial, rational polynomial, real vector, complex vector, and integer fraction types. High-level numerical capabilities include curve fitting, differentiation, differential equations, eigen values/vectors, integration, interpolation, matrix operations, root solving, and signal processing.

The I/O routines include edit dialogs/windows, display dialogs/windows, expression parsers, graphing capabilities, and file management. These routines provide support for the Macintosh interface and all numeric types in the package.

The package supports LightspeedC, and future releases will support Lightspeed Pascal and Turbo Pascal. The $125 price includes documentation, C source code, and project files for the Lightspeed environment. Reader Service No. 23.
Metaphor
P.O. Box 612
College Place, WA 99324
509-525-7120

Operating Systems
The THEOS 386 multiuser, multitasking operating system is now commercially available for 80386-based computers, from AT-class machines to super micros. THEOS 386, from **THEOS Soft-**

ware, addresses up to 4 gigabytes of memory and supports up to 128 users. Also available is THEOS C, a companion C compiler that enables Unix and DOS C programs to be recompiled and run under THEOS 386.

With the THEOS 386, a group of users can collect and send data, run a variety of programs, and communicate simultaneously by sharing one microcomputer rather than a mainframe. In addition to multiuser capabilities, the product introduces "multi-sessioning," which lets the user on the main PC console toggle between as many as 12 screens running different processes at the same time.

THEOS 386 also provides a standard application migration path, which allows users to convert their software applications for use with the THEOS 386 by recompiling. THEOS 386 also offers on-line help screens, multilingual capability, electronic mail, security passwords, I/O redirection, command pipes, and a multi-level directory structure.

The product costs $799, the single-user version is $399, and bundled development kits start at $1,599. Reader Service No. 26.

THEOS Software Corp.
1777 Botelho Dr., Ste. 360
Walnut Creek, CA 94596-5022
415-935-1118

Languages

Microsoft and **Micro Focus** have signed a strategic marketing and software development agreement with the objective of sharing product and marketing knowledge as well as producing and selling advanced Cobol application development tools and setting a standard for Cobol on OS/2 and DOS. Together the two companies will provide a Cobol development environment for professional Cobol programmers.

Micro Focus has licensed its Micro Focus Cobol/2 compiler to Microsoft for retail distribution, and Microsoft has licensed certain programming language tools and utilities to Micro Focus for retail distribution. The combined software has resulted in the Microsoft Cobol Optimizing Compiler, Version 3.0, and a new release of Micro Focus Cobol/2, Version 1.1. Reader Service No. 27.

Micro Focus Inc.
2465 E Bay Shore Rd., Ste. 400
Palo Alto, CA 94303
415-856-4161

Language Extensions I, which adds nine features to Actor has been released by **The Whitewater Group**. The features include macros, object storage, string-pattern matching, Method Browser, EditWindow enhancements, ordered dictionary, change log compression utility, literal collections, and deep copy.

The define macros introduce new syntactical forms and shortcuts, allowing conditional compiling and a variable number of arguments. With macros, you can compile one Actor program in different modes. Object storage stores whole objects onto streams or DOS files for recall at a later time. String pattern matching compares strings using wildcard characters, such as *?* and ***.

Method Browser appears as a popup window in response to a senders or implementators menu choice. It displays source code from a select group of methods. The class EditWindow has been enhanced to support horizontal scrolling and the cursor keys. With it you can edit files wider than the size of your browser and file windows.

Ordered dictionary introduces a modifiable ordering convention for dictionaries that is either chronological or sorted. Change log compression utility removes outdated code from Actor's backup log, and literal collections elimi-

STRUCTURED PROGRAMMING

Listing Three *(Listing continued, text begins on page 86.)*

```
422|   Assign (F, Filepath);
423|   {$I-}
424|   Reset (F);
425|   {$I+}
426|   IF IOResult <> 0 THEN BEGIN
427|     Writeln ('Unable to open ', Filepath);
428|     EXIT;
429|   END;
430|
431|   { Announce the program }
432|   ClrScr;
433|   IF Case_Sensitive THEN
434|     Write ('Case-sensitive')
435|   ELSE
436|     Write ('Non-case sensitive');
437|   Writeln (Heading, Filepath);
438|   Writeln;
439|
440|   { Process the file }
441|   WHILE NOT eof (F) DO BEGIN
442|     Readln (F, line);
443|     INC (LineNumber);
444|     GotoXY (1, WhereY-1); Writeln (LineNumber);  { Meter line number }
445|     Process (Line);
446|   END;
447|   Close (F);
448|   GotoXY (1, WhereY-1); Writeln (LineNumber, ' lines in file');
449|   IF CommentLevel <> 0 THEN
450|     Writeln ('Unbalanced comments detected');
451|
452|   { Alphabetize tree into non-ASCII order if case-sensitive }
453|   LineNumber := 3;
454|   IF Case_Sensitive THEN BEGIN
455|       Alphabetize (Head);
456|       Writeln (LST, 'Case-sensitive', Heading, Filepath);
457|       Writeln (LST);
458|       Report (Alpha);
459|       Count (Alpha);
460|     END
461|   ELSE BEGIN
462|       Writeln (LST, 'Non-case sensitive', Heading, Filepath);
463|       Writeln (LST);
464|       Report (Head);
465|       Count (Head);
466|     END;
467|   Writeln (LST);
468|   Writeln (LST, '-- ', NSymbols, ' symbols reported');
469|   Write (LST, Eject);
470| END.
```

End Listings

nates runtime creation of objects.

Deep copy is an object copying method that copies one object to another object recursively through all levels of data, eliminating aliasing. Language Extensions I sells for $99. Reader Service No. 28.
The Whitewater Group
Technology Innovation Center
906 University Place
Evanston, IL 60201
312-491-2370

STSC has begun shipment of the APL*PLUS II for 80386-based personal computers. The APL interpreter operates in the 32-bit protected mode of the 80386, and the memory area or "workspace" used by APL resides in extended memory.

APL*PLUS II also contains a family of language features, collectively known as *nested arrays*. The product also includes file management, full-screen editing, communications, and graphics. STSC provides a family of compatible versions of its proprietary APL*PLUS system on a variety of machines, from IBM mainframes to DEC VAX minicomputers and Unix workstations, as well as personal computers. APL*PLUS II sells for $1,500. Reader Service No. 29.
STSC Inc.
1225 E Jefferson St.
Rockville, MD 20852
301-984-5000

Version 7.0 of the APL Language Interpreter, called MultiAPL, is now available from **Spencer Organization**. MultiAPL uses various coprocessor boards based on the Motorola 680x0 microprocessor with up to 4 Mbytes of RAM available for the APL programming language under PC-DOS. Floating point support for both National's 32081 and Motorola's 6881 math coprocessors is also available. Coprocessors include a 10-MHz 68000, 12, 16, and a 20-MHz 68028.

Version 7.0 includes support for Halo 88, the newest version of the Halo Graphics Library of graphic subroutines. Other bindings include: Softcraft's Btrieve, a keyed access file system; Softcraft's version of SQL, called XQL, for use in querying Btrieve file systems; XDN, an SQL server; Flashup, an application menu and help system; and Aolmacs, a memory resident APL.68000 editor based on the Emacs programmer's editor. All optional programming language products are available from the Spencer Organization and are fully compatible with Novell Advanced Netware.

The assembly language version of APL.68000 is also available for the Atari ST, Commodore Amiga, and Apple Macintosh. Price starts at $1,495. Reader Service No. 30.
Spencer Organization Inc.
366 Kinderkamack Rd.
P.O. Box 248
Westwood, NJ 07675
201-666-6011

Parallel Processing

INMOS International has announced a VME-compatible addition to its line of TRAM (transputer module) modules and motherboards. Called the IMS BO14 module motherboard. This new product can accommodate up to 8-TRAM modules, each with its own transputer and memory.

The IMS BO14 is compatible with VMEbus Specification rev C.1 and is a standard depth, double-height card 6U (height) by 160 mm (width). The BO14

OF INTEREST

(continued from page 139)

extends the range of buses with which the transputer now interfaces to include VME, VAX, Sun Macintosh, and the IBM PC. The product sells for $2,040. Reader Service No. 31.
INMOS Corp.
P.O. Box 16000
Colorado Springs, CO 80935
719-630-4215

Hecht-Nielsen Neurocomputers has released the Anza Plus/VME, a neurocomputing coprocessor board for Sun workstations that is supplied with software that facilitates the integration of neurocomputing into C programs. With this product, users can apply neural network technology to image processing, high-speed industrial inspection, signal procession, and process control.

Equipped with 10 Mbytes of onboard RAM, the Anza Plus/VME is a single-board coprocessor that installs in any Sun 3 VME slot. All neurocomputing functions are performed by this product, and it can implement neural networks with any combination of up to 2,500,000 processing elements and interconnections. The coprocessor can update the network at a rate of 10,000,000 interconnects/sec. (IPS).

Costing $25,000, the Anza Plus/VME is supplied with release 2.1 of HNC's neural network software system, operating under the Sun Unix operating system. Reader Service No. 32.
HNC
5501 Oberlin Dr.
San Diego, CA 92121
619-546-8877

Miscellaneous

Cogent Software has announced the availability of HyperBase, an application development tool for the IBM PC. Combining Hypertext, interactive graphics, and the artificial intelligence programming language Prolog, HyperBase permits users to develop and deliver intelligent documents on an IBM PC.

Authors and readers of documents do not have to be familiar with Prolog to develop or use applications, yet they can produce documents that contain complete expert systems attached to both graphics and text. Furthermore, intelligent Hypertext links may be constructed that aid a reader in using the document and that modify themselves according to the reader's skill level.

HyperBase comes with a toolkit for capturing images directly from the computer screen and incorporating them into documents. A built-in Prolog interpreter supplies the intelligence engine. Intelligent HyperBase documents may be created by annotating pre-existing documents in a text editor.

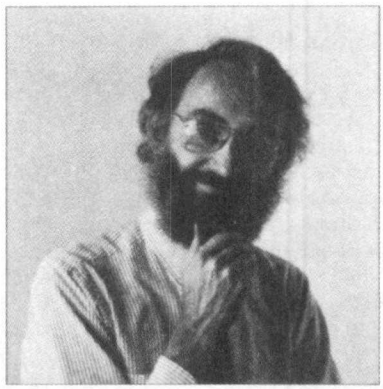
HyperBase requires an IBM PC, AT, PS/2, or compatible with at least 384K free memory and DOS 2.1 or later. A hard disk and graphics display are recommended, and mouse support is provided.

HyperBase is available in two versions, a $99 personal version and a $249 developer system, which permits the creation and distribution of standalone applications. HyperBase is compatible with the Cogent Prolog compiler (available separately for $199), and code produced with the compiler may be attached dynamically to HyperBase documents. Reader Service No. 34.
Cogent Software Ltd.
21 William J. Heights
Framingham, MA 01701
508-875-6553

DDJ

SWAINE'S FLAMES

The Macintosh Fan



There's this essay I'd like to write someday, all about how corporate decisions are made and how they are subsequently assessed; and about the role of luck, fortuitous timing, and amazing consumer tolerance in the success of Apple Computer.

The title of the essay would have to be "The Apple Fan," with a double meaning and a twist of irony in each of the meanings.

The essay would tell the story of two products: the 128K Macintosh, cute and cuddly, the triumph of concept and style over performance; and the power Mac, a composite of the Plus, SE, and II.

The absence of a fan in the original Mac would serve as a model for Steve Jobs' insistence on keeping the Mac from looking, sounding, or feeling like a computer, a goal that could only be attained by keeping it from being fully a computer. Thus the fan would also symbolize the serious performance limitations of the early Macintosh.

The essay would portray Mac users symbolically demanding "We want a fan!" and show that the users could then hardly object when the more powerful, fan-equipped Mac they had demanded turned out to be less friendly, less cuddly than the family pet they had initially taken to their hearts. After all, power comes at a cost. And the essay would show Mac SE and II users accepting their computers as being in every sense Macintoshes, but with the power of a "real computer."

Could Apple have succeeded as it has if it had delivered a more powerful but less conceptually pure computer in 1984? Or even if it had delivered the Mac II back then? The essay would argue that it could not, that the sequence of "toy" Mac and "real computer" Mac was the key to Apple's current success, because it opened a market and created a specific demand for a more powerful Mac —a machine that only Apple could produce and that Apple could easily produce.

And the essay would argue with malicious delight that the sequence was nobody's decision but merely the accidental result of the successive ascendencies of different factions within the company. A historical accident of the "hardware wars."

One of these days, I'll get around to writing it.

Thank You, Thank You, Thank You

Even *Time* magazine is warning us about computer viruses. That's all we needed; now the obnoxious pranksters can aspire to make the cover of *Time*. I deplore the actions of the virus creators, but the real dangers they pose are one that, if encouraged, the practice might become a fad, and two that the general public might become so alarmed as to support crippling of the hardware or operating system or legislation that threatens our civil liberties. *Time* magazine and others who promote virus hysteria encourage both these possibilities.

One of the most cogent thinkers on civil liberties in the information age is Dean Gengle, who has graced these pages and who reminded me recently about the real danger of computer viruses. I thank him for that, and I thank George Bush for reminding me to renew my ACLU membership.

And while I'm thanking people, let me thank the editor of a short-lived but much-appreciated programming magazine, *Turbo Technix*. Jeff Duntemann is one of the best writers and editors in the industry. In the October issue of the newsletter he sent to his contributors, he announced that the September/October issue of the magazine would be the last, in the interest of Borland's profitability picture. It is characteristic of Jeff that this newsletter, with an audience of a few dozen, was better written than most computer magazines.

Exploratory Data Analysis

Do you get the DAK catalog too? "Forget high stereo prices. Forget changing channels by hand." There's a man who gets his money's worth out of his word processor. Anyway, forget *computer viruses*. The real menace is exploratory data analysis.

Exploratory data analysis is a legitimate statistical tool, but imagining it in the hands of the average PC user is scary. Here's why:

Using statistical techniques requires recognizing the underlying assumptions. Some statistical techniques are robust to violations of assumptions, and can be applied loosely; others are less forgiving. Few people who work with conventional statistical techniques know the assumptions.

Exploratory data analysis invites a whole different category of assumption violation: the confusion of the context of discovery and the context of justification. Data used to come up with a hypothesis is not valid to use in supporting the hypothesis. Obviously. Just watch that obvious assumption be violated right and left as naive users play around with exploratory data analysis.

Michael Swaine

Michael Swaine
editor-at-large

Dr. Dobb's Macintosh Special

Summer, 1988

THE MACINTOSH TECH JOURNAL

Dr. Dobb's MACINTOSH SPECIAL™

Display until
September 30, 1988
3.95 (4.95 CANADA)

RESOURCES FOR THE MAC PROGRAMMER

INSIDE THE MACINTOSH PROGRAMMER'S WORKSHOP

A/UX: UNIX FOR THE REST OF US?

WRITING HYPERCARD XCMDs

0 38351 17742 3 85

SUMMER ISSUE

CONTENTS

THE MAC PROGRAMMING ENVIRONMENT

A MAC PROGRAMMER'S RESOURCE KIT *by Anthony Meadow*

8 Tony presents an overview of the plentiful (albeit, occasionally difficult to locate) resources available to the Mac programmer. The guided tour includes: development systems, software, organizations, training, user groups, and publications.

THE MACINTOSH PROGRAMMER'S WORKSHOP *by Dan Allen*

20 Dan Allen is your guide on a tour of the Macintosh Programmer's Workshop (MPW). Dan points out the benefits of MPW's integrated programming environment, and examines a selection of its powerful tools and resources—including integrated tools.

35 **RESOURCE GUIDE** *by DDJ Staff*
A selection of programming products, services, and publications.

41 **BIBLIOGRAPHY** *by DDJ Staff*

THE MAC II

APPLE'S A/UX: UNIX FOR THE REST OF US? *by Anthony Meadow*

44 Tony explores the Mac II platform for developing Unix system applications. He delves into the critical issues of compatibility and functionality with an eye toward the special problems/opportunities of programming on and for the Macintosh interface. Tony also includes strategies for combined Mac OS and A/UX development, and a wish list for future releases of A/UX.

46 **RESOURCE GUIDE** *by DDJ Staff*
A selection of A/UX products, services, and publications.

55 **BIBLIOGRAPHY** *by DDJ Staff*

HYPERTALK

INTRO TO HYPERCARD PROGRAMMING *by Dan Allen*

56 One of HyperTalk's principle programmers, Dan Allen explains how to expand the power of HyperCard through external procedures to handle such tasks as: custom device drivers, access to the Mac OS and ToolBox routines, and speeding up the execution of time-critical code. Dan includes a sample XCMD written in MPW C.

63 **RESOURCE GUIDE** *by DDJ Staff*
A selection of HyperCard/HyperTalk language products, services, and publications.

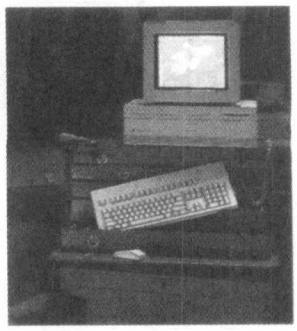
64 **BIBLIOGRAPHY** *by DDJ Staff*

FORUM

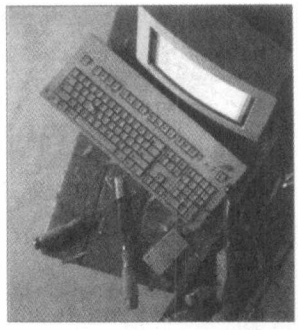
6 **THE REBIRTH OF THE MAC** *by Randall P. Sutherland*

75 **VENDOR DIRECTORY** *compiled by the DDJ Staff*

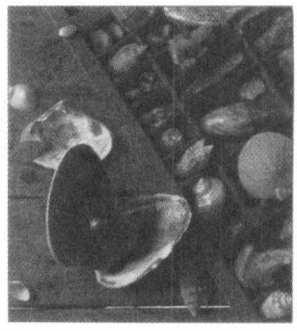
80 **THE DEATH OF THE MAC** *by Michael Swaine*



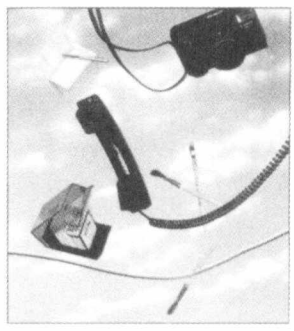
8



20



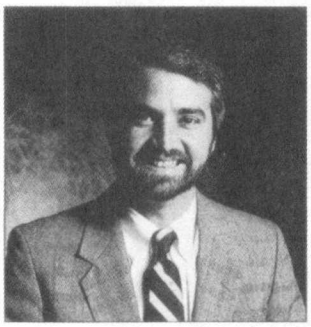
44



56

The editors wish to thank Thea Grigsby of SuperMac Software (Mountain View, CA), for her technical assistance with the cover art and for the contribution of PixelPaint, the program used to produce the colorful image seen on the Mac II monitor. Also, special thanks to Michael Carr, our cover photographer, whose willingness to work long hours is cheerfully acknowledged and appreciated.

THE REBIRTH OF THE MACINTOSH



Randy Sutherland is a research analyst at Dataquest, a San Jose, CA, market research firm, source of this Apple sales projection.

You know the Apple story: Steve Wozniak builds a nifty plaything, Steve Jobs turns a project into a product, VisiCalc turns it into a business tool, and the Apple of Woz's eye invades corporations through backdoors and budget loopholes. Apple Computer bootstraps itself overnight into the Fortune 500, whereupon IBM wakes up, blesses Microsoft, and calls all the shots for the next two years.

That was the world of personal computing when the Apple Macintosh was born in January 1984. Fitted with a user interface that was a generation beyond anything else commercially available, the 128K machine was nevertheless hopelessly bogged down by that very interface. The learning curve for developers was steep, and Apple had bundled powerful applications with the machine, which discouraged development. Third-party software was slow in coming.

Then in 1987, the Mac was born again. Now Apple profits are soaring, and even Mitch Kapor is developing software for the Mac. What is behind the Mac's rebirth?

One answer is power. Hard disks, more memory, slots, and a decent local-area network all help to put a real structure behind the snazzy facade. DOS or Unix software developers who had never considered developing Mac applications are now taking a second look at the computer that has evolved from an appliance to a full-blown development platform that supports a lucrative third-party industry.

But another answer involves the balance between flexibility and standards.

The open design of the Apple II and the IBM PC provided programmers with a canvas for their creativity. But creativity in the design of user interfaces forces the user to thread a new maze with each new application. With the Mac, Apple enforced a uniform user interface, which made the programmer's job harder. But it made the user's job easier, and in the long run that proved more important. As the number of applications increased and the number of interfaces to learn didn't, the Mac grew in popularity.

But IBM has not been napping.

Whatever may have motivated Apple's suit against Hewlett-Packard and Microsoft over Windows 2.03, the controversy has served to emphasize that Windows and the OS/2 Presentation Manager are an attempt to bring the benefits of the Mac user interface to IBM's turf. A question suggests itself. OS/2 Extended Edition is clearly the way to go for anyone needing transparent access to IBM mainframes, but why should others wait for something like a Mac when they can have a real Mac right now?

As IBM's personal computer strategy swings in a proprietary direction, Apple is moving toward participation in industry standards. History lesson: Digital Equipment ignored Sun Microsystems' coexist-and-conquer strategy until Sun could no longer be blocked.

Under John Sculley, Apple is making one right move after another:

- The recent Apple/DEC agreement to agree on standards encourages a unified Mac/VAX alternative to IBM.
- Apple's investment in Sybase implies Mac participation in distributed data processing environments designed around SQL.
- With A/UX, Apple is again integrating the Mac with industry standards.

The power and the careful adoption of industry standards are paying off. For every two PS/2s (Models 50 to 80) that IBM ships this year, Apple will ship a Mac.

RP Sutherland

A MAC PROGRAMMER'S RESOURCE KIT

by Anthony Meadow

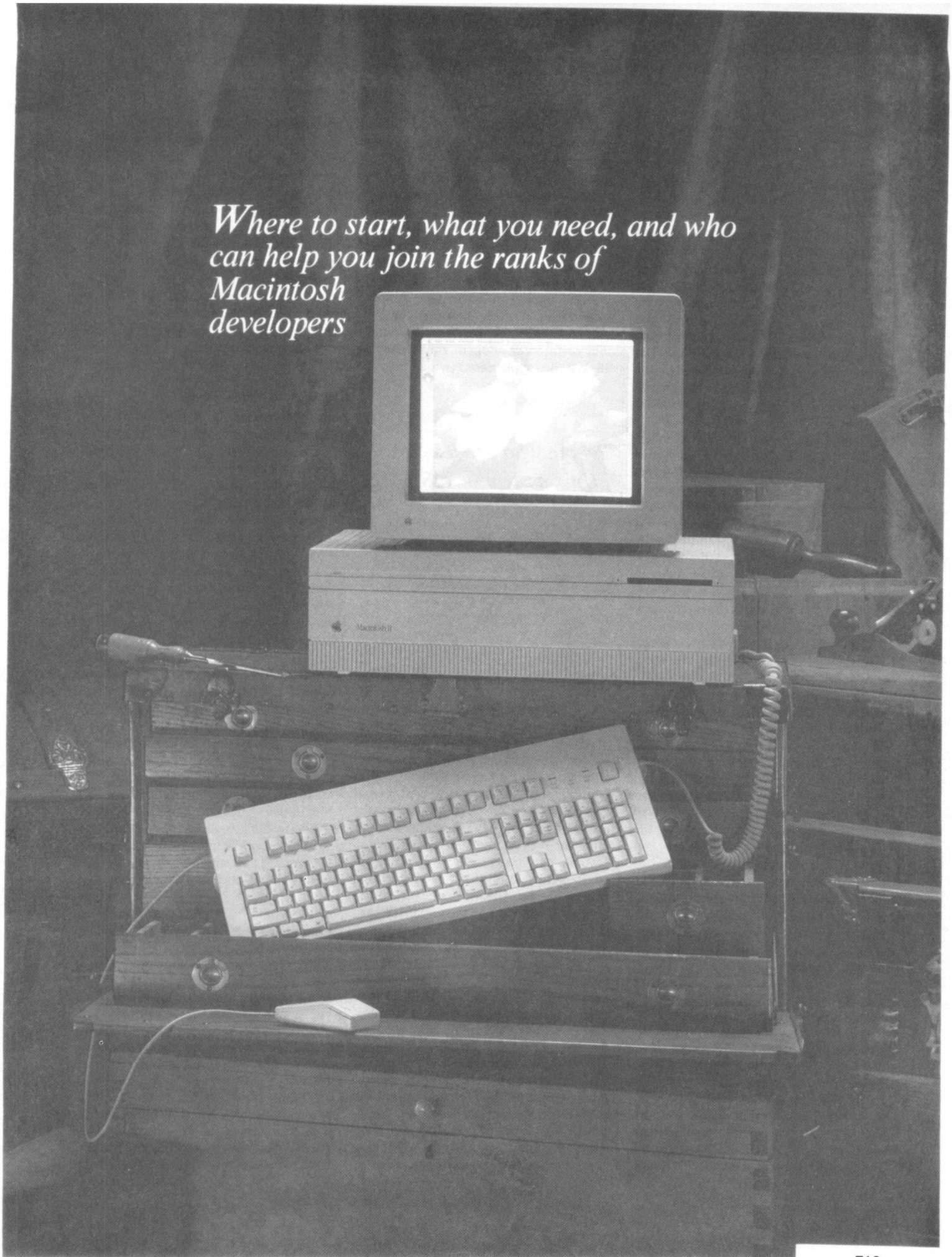
High user expectations put a lot of pressure on any new software program—especially one written for the Macintosh. Not only must a new Macintosh program have all the required and anticipated functions, but is must also be better (i.e., more powerful, have more features, and be easier to use) than those that have come before. It must accomplish this with novelty and innovation while meshing seamlessly with the existing body of programs that collectively comprise what is known as "the look and feel of the Mac." This is both a blessing and a curse.

The Macintosh operating system includes a powerful battery of procedures and resources to draw upon for designing the user interface and screen displays. These powerful high-level features were provided by the Macintosh system software designers to make it easier for programmers to meet the Macintosh look and feel criteria. Now for the bad news. . .

Macintosh programs are event-driven; this means that a program must continuously poll for events (such as a mouse event, keyboard input, or an interrupt from an external device) and immediately respond with the appropriate function. Thus, all functions must be available at all times. Instead of having the luxury of dividing an application into modules with specific functions available in certain situations, a programmer must provided the user with most, it not all, of an application's functionality at the drop of an event. That's not an easy chore.

The up side: Apple and others aren't blind to this issue. Because the transition to programming for the Macintosh is likely to include a steep learning curve, Apple and other affiliated groups do offer assistance in the form of technical training, specialized software, and other support. In this article, Tony Meadow, a Macintosh entrepreneur in his own right, shares his knowledge about what's available from and to the Macintosh development community and where you can go to find it.

Then all you have to worry about is the simple stuff like finding a need, and filling it, while one-upping everyone else in the process. Best of luck.—ed.


Where to start, what you need, and who can help you join the ranks of Macintosh developers

(continued from page 8)

Help Wanted

Last year the world ran low on yet another vital resource—Macintosh programmers. After the Macintosh II arrived, a lot of companies that thought the Macintosh would never be more than a toy decided that just maybe they should reevaluate their position. Then they got scared. Unfortunately, the Macintosh is a different animal, and programming a Macintosh application takes both specialized knowledge and ability to play by Apple's rules. The result: There's a shortage of Macintosh programmers. So, how come I know so much about it?

I serve as the director of MacSEF (the Macintosh Special Interest Group of the Software Entrepreneurs' Forum). This group is made up of both current and aspiring Macintosh developers who are located in the San Francisco Bay Area. About a hundred people come to its monthly meetings. And the question I'm asked most often these days is, "Exactly how do I become a Macintosh developer?" Well folks, this article is my attempt to answer that question. In it, you'll find lists of every resource I could think of to help you become a Macintosh programmer.

Apple Computer's Certified Developer Program

The most obvious (well, it should be, anyway) and important place to begin is with Apple Computer. It does more than any other company that I'm aware of to encourage people to write software for its machines. Apple does this through its Certified Developer Program. If you apply (and are subsequently accepted), Apple will provide you with buckets of resources. Among these resources are discounts on machines, the *Outside Apple* newsletter, special mailings, and access to AppleLink (Apple's internal bulletin-board and e-mail system). You will also receive *Technical Notes*, which provide details of recent changes to the Macintosh operating system. Apple also holds topical conferences for developers; both technical and marketing people who have attended those held in the past have found the conferences very useful.

It's particularly important that you sign up for the AppleLink electronic-mail network, which makes communicating with almost everyone at Apple very easy. There are quite a few people who, though notorious for playing telephone tag for weeks, will respond promptly if you send them an AppleLink e-mail message. AppleLink costs more than some other networks—there's a minimum fee of $25 per month. On AppleLink, you're billed for connect time with different rates for daytime or off-hour use.

Don't think that acceptance into the program is routine, however. It will take some work on your part to qualify as a certified developer. You'll need to fill out an application (which can be obtained by calling the Developer Program's group at Apple at 408-973-4897). Developers who actually have products shipping (on any development platform) usually have an easier time getting accepted, but start-up companies are frequently accepted as well. Once you are into the program, you'll find that Apple provides support and encouragement like no other computer manufacturer.

People who are used to the Unix system or MS-DOS should have very little culture shock with MPW.

After you are certified, you can contact technical support using AppleLink. (There is no technical-support phone number available for those outside Apple.) Even if you're not certified, you can still ask technical-support questions via MCI Mail. MCI Mail costs $18 per year and you will be billed for each message that you send—but not for connect time (not yet, anyway). Send messages to MACDTS and you should get a reply within a day or so.

There's another group within Apple that is very helpful to developers—the Apple Evangelists. Their mission has been to go to companies and encourage the development of software and hardware products for all Apple's computers. Each evangelist has an area of specialization, such as business software or K–12 educational software. The evangelists act as advocates for third parties within Apple. They are also responsible for sending developers early versions of Apple's upcoming products.

Finally, there is the third-party marketing group at Apple. It helps companies by including them in press-release packets, inviting them to demonstrate their products at Apple's booth at various trade shows, and so on. Although it doesn't (and can't) help everyone, small developers as well as large developers are invited to participate.

APDA—Apple Programmer's and Developer's Association

Apple has helped set up a mail-order house called APDA, the Apple Programmer's and Developer's Association. Apple publishes almost all of its developer tools and technical documentation through APDA. Often they come out in draft form initially, which usually means they don't include the final artwork or indexes, but it does mean you can get tools and documentation a lot earlier than if you had to wait for the final versions, which are often published by Addison-Wesley. Until the final releases are available, APDA is the only source for many tools and most documentation. It costs $20 per year to join APDA ($25 in Canada and Mexico and $35 elsewhere). You'll receive a quarterly bulletin that lists all available products, including many third-party developer tools.

APDA is the only source for the Macintosh Programmer's Workshop (MPW), Apple's development environment for the Macintosh. MPW supports C, Pascal, and assembly language, as well as Object Pascal (an object-oriented language based on a few extensions to the Pascal language) and C++ (available later this year). A Unix-like shell with an integrated editor comes with MPW. The shell provides most of what's great about the various Unix shells (scripts, powerful command language, and so forth) with a very Macintosh-like interface. People who are used to the Unix system or MS-DOS should have very little culture shock with MPW.

There are all sorts of other things available besides MPW, such as technical documentation on AppleTalk, the LaserWriter family, A/UX, and HyperCard. Subscriptions to the *Technical*

(continued from page 10)
Notes for the Macintosh are available, as well as all the previous issues. The *Technical Notes* provide sample code for some of the trickier aspects of the operating system, programming guidelines, some file formats, and so on. They are essential for anyone developing commercial software for the Mac. APDA also carries many third-party products, including most of those listed in this article.

Good Reading

One of your next stops should be a good computer bookstore. In the Bay Area, there are several good ones—Computer Literacy and Stacey's are two good chains that seem to have most computer books. If you aren't located in an area with a store that carries the latest technical books, you can order them from Computer Literacy and have items shipped to you.

Addison-Wesley publishes a series of books that have been written by various groups at Apple Computer. The most important of these is *Inside Macintosh*, which consists of five volumes. There is now a sixth book, which is a complete index to all five volumes, called *Inside Macintosh X-Ref*. The first three volumes describe the original Macintosh operating system, which is commonly referred to as the Macintosh ROM. The fourth volume describes the additions that were made to the operating system when the Macintosh Plus was released. The fifth volume describes the changes made to the operating system when the Macintosh SE and Macintosh II were released. Yes, you'll need all five volumes and the combined index comes in pretty handy as well. *Inside Macintosh* is not easy reading, but it is mandatory. You'll end up reading the series from cover to cover several times. (There are other books that are, perhaps, better to start with, some of which are listed later.)

Another important book in Addison-Wesley's Apple series is *Human Interface Guidelines*. In the Macintosh world, users pay more attention to the user interfaces of applications than do users of other machines. In fact, one of the primary reasons why the Macintosh has succeeded is the interface guidelines. In virtually all Macintosh applications, cutting and copying of text and graphics, the behavior of windows and dialog boxes, and quitting the application are all done in the same way. This makes it much easier on the users, and better for programmers too, because users can transfer much of their knowledge from one program to another. The average Macintosh user regularly employs about six applications, whereas the average MS-DOS user employs perhaps three.

Other volumes in Addison-Wesley's Apple series include *Technical Introduction to the Macintosh Family*, *Programmer's Introduction to the Macintosh Family*, and *Designing Cards and Drivers for Macintosh II and Macintosh SE*. There will undoubtedly be other volumes in this series in the future.

For those just beginning, the two-volume set, *Macintosh Revealed*, by Stephen Chernicoff, is a good place to start. The set provides an easy path to learning the operating system because it starts with the most important managers (parts of the operating system) and continues to examine the system in more and more detail. There are quite a few managers that are not mentioned in this set, but all the important ones are. Volume 1 is called *Macintosh Revealed: Unlocking the Toolbox*, and volume 2 is called *Macintosh Revealed: Programming with the Toolbox*.

Other good books are two by Scott Knaster, who used to manage the Technical Support group at Apple. They are called *How to Write Macintosh Software* and *Macintosh Programming Secrets*. These books are for programmers who already know the basics of the Macintosh OS and need to understand more about its inner workings. These books should be on your must-read list; even the most experienced Macintosh programmer will get something out of them.

Apple is quite interested in object-oriented programming. Technology explorers at Xerox PARC, such as Larry Tesler and Dan Ingalls, have infected the Apple development crowd with the fundamental ideas of object-oriented programming, and as a result, Apple now supports an object-oriented, language called Object Pascal. Object-oriented approaches are only just beginning to make themselves felt in the Macintosh marketplace.

Pascal and C are the two languages most often used in the Macintosh world.

For example, MacApp is an application toolkit (available from APDA) that was written in Object Pascal by Apple programmers. It can serve as a foundation and framework on which to build your own program. There is also a version of C++ that works with MacApp objects in the works—it should be out sometime this year. Finally, Kurt Schmucker, who's now working for Apple, has written a book called *Object-Oriented Programming for the Macintosh*, a second edition of which should be available later this year.

There aren't a lot of magazines oriented toward programmers and developers in the Macintosh world. However, there is one magazine, *MacTutor*, which has been around for almost three years now, that regularly prints articles with a lot of source code. There's also a newsletter called *Connections*, which covers developments in the AppleTalk arena. Their articles are technical and oriented toward developers and those who use large, complex networks.

Programming Languages

Pascal and C are the two languages most often used in the Macintosh world. Pascal is common for historical reasons. Apple used the UCSD Pascal system on

(continued from page 15)
the Apple II and then the Apple III. When the Lisa (the predecessor of the Macintosh) was being designed, the UCSD Pascal system was modified and ported over and eventually became the Lisa Workshop. In the early days, the only way to develop Macintosh software was with the Lisa and the Lisa Workshop. Since then, C has become more popular. Today, there's probably a 50–50 split between Pascal and C among developers. Although there is always some work that's done in assembly language, the majority of all Macintosh products are done in either C or Pascal.

There are two basic flavors of development environments in the Macintosh world. In one corner, you have development systems with very Mac-like user interfaces. The best known examples of these are Lightspeed C and Lightspeed Pascal. Both are tightly integrated environments that include an editor, compiler, linker, and *make* facilities (Lightspeed Pascal also includes a source-level debugger). Both are very fast and easy to learn and use. Their disadvantages are that they aren't extensible and it's a bit of work to develop software in a group.

The other approach is to have a more Unix-like interface. The Macintosh Programmer's Workshop (MPW), mentioned earlier, is the best known example of this approach. MPW still has a Macintosh-style interface even though it has a Unix-like shell, and the environment is easy to extend. It's also easy to port well-written Unix or MS-DOS utilities over to MPW because there are standard input/output libraries for both C and Pascal. The linker allows the use of Pascal, C, and assembly language in a single application.

There are several third parties that offer Modula-2 and Fortran compilers that run under MPW. MPW's disadvantage is speed; it is noticeably slower than the Lightspeed products.

There are several additional vendors of C and Pascal: Manx Software, Borland International, TML Systems, and Consulair number among them. It does seem, however, that MPW and Lightspeed are the most widely used.

As alluded to before, object-oriented programming is becoming increasingly popular in the Macintosh world. It is frequently used within Apple; most of the programs used to demonstrate the Macintosh II when it was first released were written in MacApp. MacApp, the "Expandable Macintosh Application" (written in Object Pascal), is a set of

Stepping Stones

General Information

Apple Computer Inc.
20525 Mariani Ave.
Cupertino, CA 95014
Main number: 408-996-1010
Developer Program's group:
408-973-4897
(Mailstop 51-W)
Evangelists: 408-973-3289
(Mailstop 51-AB)
Developer tech support:
no phone number
(Mailstop 51-T)
MCI Mail: to open an account
800-444-6245

APDA (Apple Programmer's and Developer's Association)
MPW, MacApp, *Tech Notes*, etc.
290 S.W. 43rd St.
Renton, WA 98055
206-251-6548
AppleLink: APDA
MCI Mail: APDA
CompuServe: 73527,27

Good Reading

Computer Literacy Bookstore
2590 N. First St.
San Jose, CA 95131
408-435-1118

Connections
Newsletter covering developments in the AppleTalk world
P.O. Box 5894
Fullerton, CA 92635
714-738-1492

MacTutor
Technical magazine
P.O. Box 400
Placentia, CA 92670
714-630-3730

Programming Languages

APDA
MPW (Macintosh Programmer's Workshop)
see above

Coral Software Corp.
Allegro Common LISP
P.O. Box 307
Cambridge, MA 02142
617-547-2662

ParcPlace Systems
Smalltalk-80
2400 Geng Rd.
Palo Alto, CA 94303
415-859-1000

Semantic Microsystems
MacScheme+ToolSmith
4470 S.W. Hall St., Ste. 340
Beaverton, OR 97005
503-643-4539

Think Technologies
Division of Symantec
Lightspeed C and Pascal
135 South Rd.
Bedford, MA 01730
617-275-4800

Other Development Tools

ALSoft
MacExpress
P.O. Box 927
Spring, TX 77383
713-353-4090

Faircom Systems
C-Tree and R-Tree
4006 W. Broadway
Columbia, MO 65203
314-445-6833

Invention Software Corp.
Programmer's Extender Series
P.O. Box 3168
Ann Arbor, MI 48106
313-996-8108

Raima Corp.
db'Vista and db'Query
3055 112th Ave. NE
Bellevue, WA 98004
206-828-4636

objects that implement all the standard behaviors of a Macintosh application. The programmer writes code for functions that are application-specific and MacApp handles the rest. Like MPW, it is available through APDA.

For Lisp fiends, there are two excellent environments. Coral Software's Allegro Common Lisp is a wonderful implementation with a well-conceived package that includes an Emacs-like editor and a source-level debugger. There is also a foreign-language (C, Pascal, or assembly language) interface. Object Lisp is supported now and CLOS (Common Lisp Object System) is coming soon. Although it can't produce standalone (double-clickable) Macintosh applications yet, this feature should be available soon. Only 1 Mbyte of RAM is required, so it's not a memory hog.

MacScheme+ToolSmith from Semantic Microsystems is a nice implementation of Scheme, the C of the Lisp world. It also has a Macintosh user interface and a source-level debugger. There is support for calling assembly-language routines but not C or Pascal. There's also an Application Builder, which produces double-clickable applications.

At this time, there isn't a Smalltalk that has a Macintosh user interface or that can create applications with a Macintosh interface. ParcPlace Systems (a spin-off of most of the Smalltalk wizards from Xerox PARC) does have a well-supported, well-documented Smalltalk for the Macintosh. It has the traditional Smalltalk user interface, though, so it isn't of interest to those who want to develop Macintosh applications. Digitalk is working on versions of its Smalltalk V for both the Macintosh II and SE. And finally, Apple has an unsupported, experimental version of Smalltalk (Version 0.4), which is available through APDA. Apple is also working on a much more elegant version with a Macintosh interface, although it's likely that it won't be available until late this year (if we're lucky).

Other Development Tools

One nice tool that saves a lot of time is the Programmer's Online Companion, available from Addison-Wesley. It provides the interface definitions to most of the Macintosh OS calls. Experienced programmers spend more time with the five volumes of *Inside Macintosh* than anything else, so this product pays for itself very quickly.

I'm aware of only four database or file manager toolkits available for the Macintosh. Inside-Out, from Shana En-

Shana Enterprises Inc.
Inside-Out relational database
#200, 9704 - 54 Ave.
Edmonton, Alberta T6E 0A9, Canada
403-438-6548

SmethersBarnes, Technology Division
Prototyper
P.O. Box 639
Portland, OR 97207
503-274-2810

TML Systems
TML Database Toolkit
4241 Baymeadows Rd., Ste. 23
Jacksonville, FL 32217
904-636-8592

Debuggers

Icom Simulations
TMON
648 S. Wheeling Rd.
Wheeling, IL 60090
312-520-4440

Jasik Designs
The Debugger
343 Trenton Wy.
Menlo Park, CA 94025
415-322-1386

Other Organizations

ATDA (AppleTalk Developers Assoc.)
c/o Guy Mariande
Tangent Technologies
5720 Peachtree Pkwy., Ste. 100
Norcross, GA 30092
404-662-0366

MacTechGroup
Boston Computer Society
48 Grove St.
Somerville, MA 02144
617-625-7080

DC MPW Developer's Group
c/o Michael F. Hartman
Open Cluster Software
8116 15th Ave., #201
Adelphi, MD 20783
301-445-1583

DEDA (Desktop Engineering Developers Assoc.)
c/o Bill Stanley
The Computer Shoppe
615 Guilford-Jamestown Rd.
P.O. Box 18344
Greensboro, NC 27419
919-299-4843

MacSEF (Special Interest Group of the Software Entrepreneurs' Forum)
P.O. Box 61031
Palo Alto, CA 94306
415-854-7219
Contact: Tony Meadow
415-644-9400

MADA (MacApp Developers Assoc.)
P.O. Box 23
Everett, WA 98206-0023
AppleLink: X0591
MCI Mail: Carl Nelson

MacTechGroup (Part of Boston Computer Society)
Boston Computer Society
One Center Plaza
Boston, MA 02108
617-227-7986
Contacts: Maynard Chen
617-492-7463
Brian DeLacey
617-876-0165

Getting a Head Start

Bear River Institute Inc.
P.O. Box 1900
Berkeley, CA 94701
415-644-0555

Personal Concepts Inc.
635 Wellsbury Wy.
Palo Alto, CA 94306
415-494-6763

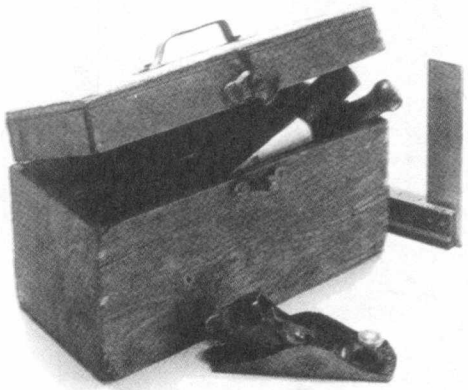
(continued from page 17)
terprises, is the most advanced. It is a relational database system that you link into your code. Pascal, MacApp (Object Pascal), and C are supported. A multiuser version that works with AppleTalk will be available in the near future. Source code is not available, and there is a licensing fee.

C-Tree, a B-tree package, and R-Tree, a report writer, are available from Faircom Systems. Both have been ported to the Macintosh and run on a wide variety of machines. Source code for both is available and there is no licensing fee.

The TML Database Toolkit, available from TML Systems, is an ISAM package written in TML Pascal. Presumably, it could be easily moved to other Macintosh Pascals. Also available are db'Vista, a network model database, and db'Query, a query package, from Raima Corporation. Source code is available.

MacExpress, from ALSoft, and the Programmer's Extender series, from Invention Software Corp., are toolkits that can give you a running start in developing Macintosh applications. They provide the basic behavior of a standard Macintosh application so that you (hopefully) only have to write the application-specific code. These kits can be used with C or Pascal. Source code is available for the Programmer's Extender series. Both kits attempt to provide the advantages of an object-oriented toolkit (such as MacApp) in traditional procedural languages. They can only be partially successful in this attempt, and it is my impression that MacExpress is somewhat the better of the two.

Prototyper, from SmethersBarnes, as its name suggests, is an interface design tool for creating such objects as menus, windows, dialog boxes, alerts, and icons. It also generates complete Pascal source code and resources for the Lightspeed, MPW, Turbo, or TML compilers. No distribution or licensing fees are charged.



Debuggers

Three debuggers are available for the Macintosh. Not one of them is sufficient for all needs, so you'll probably end up using at least two of them. Unfortunately, there's only one source-level debugger available on the Macintosh now, and that is the integrated debugger in Lightspeed Pascal. I've heard rumors of two or three source-level debuggers under development, but until they're available, you'll have to learn (at least) a little assembly language.

There's a way to get a head start on learning the Macintosh operating system: Take a class.

MacsBug is the debugger supported by Apple as part of MPW. Although it is derived from the original version developed by Motorola, it's been rewritten several times.

TMON was the second debugger for the Macintosh. It has a window-based user interface, but it does not use the Macintosh user interface. Because it's window-based, you can look at several different things (registers, code, data areas, and so on) simultaneously. There are some capabilities beyond MacsBug, including the ability to extend it by writing a user area. It also supports trap discipline, which is a series of patches installed in front of system calls to check for common errors, such as invalid handles (pointers to pointers), invalid strings, and so on. The latest version (2.8) runs on a Macintosh II but does not understand the 68020 specific, 68881 FPU, and 68851 PMMU instruction sets.

The Debugger, from Jasik Designs, is the latest debugger for the Macintosh. It has a Macintosh user interface and provides some features that neither MacsBug nor TMON have. Among other features, it knows the symbolic names of standard Macintosh OS globals and data structures. You can also easily teach it about your application-specific data structures.

Other Organizations

A useful way to share information about the Macintosh is through developers' groups. I'm only aware of three in the U.S. These are MacSEF (the Macintosh Special Interest Group of the Software Entrepreneurs' Forum) in the San Francisco Bay Area (of which I'm the director), the MacTechGroup (of the Boston Computer Society), and the DC MPW Developer's Group (Washington, D.C.).

These groups are composed mainly of programmers and developers who are writing code for the Macintosh, not end-users. They provide developers with an opportunity to share technical information, meet other people who are also excited about the Macintosh, perhaps make contacts useful in the business side of things, and hear interesting speakers. At MacSEF, for example, speakers have included Scott Knaster, Erich Ringewald, Phil Goldman (who wrote MultiFinder), Larry Rosenstein (who wrote a lot of MacApp), and Bill Campbell (president of Claris Corp.—Apple's application software spin-off).

There are several national organizations of Macintosh programmers and developers that were organized to focus on a specific interest.

The AppleTalk Developers Association (ATDA) was organized a couple of years ago by several companies developing products that work with AppleTalk, Apple's LAN. AppleTalk, which many people outside the Macintosh world don't know a lot about, is a very sophisticated suite of protocols. Apple provides two implementations of it: LocalTalk (built into every Macintosh, runs at 230K/second, and is quite inexpensive and easy to install) and EtherTalk (based on top of Ethernet, runs at 10 Mbytes/second, and is more expensive to install). The organization promotes cooperation among members, including both the technical and marketing angles.

The Desktop Engineering Developers Association (DEDA) was organized by several companies that have products in the engineering/scientific marketplace. They also promote cooperation among

interested parties.

The MacApp Developers Association (MADA) is a users' group for those interested in MacApp, the Expandable Macintosh Application, written in Object Pascal. It has meetings twice a year at the San Francisco Macworld Expo show (each January) and the Boston Macworld show (each August). There are five disks of source code and various tools that the group has put together so far. The MacApp development group at Apple attends each meeting and talks with programmers who use MacApp. The members listen and actually make changes to MacApp based on suggestions that come up at these meetings. MADA is therefore a very worthwhile group. Anyone interested in object-oriented programming on the Macintosh should join.

Getting a Head Start

There's a way to get a head start on learning the Macintosh operating system: Take a class. Classes are offered by three companies: Apple Computer, Personal Concepts, and Bear River Institute (of which I'm president and in which I obviously have a vested interest).

Apple Computer offers four courses: *Using Macintosh* Programmer's Workshop (1 day), MacApp and Object-Oriented Programming (4 days), Introduction to Macintosh Programming (1 day), and Developing a Macintosh Program (1 day). These courses are available primarily to certified developers, and there is often a waiting list to get into a class.

Personal Concepts and Bear River Institute offer classes primarily on an in-house basis, but classes are sometimes available to the general public. Both companies offer courses designed to complement those offered by Apple.

The classes available through Personal Concepts are Programming the Macintosh II (1- and 4-day versions) and Macintosh Programming—An Introduction (1- and 4-day versions).

Classes available from Bear River Institute are Advanced Macintosh Programming (4 days), Testing Software on the Macintosh (2 days), Designing a Macintosh Application (2 days), A Technical Introduction to AppleTalk (2 days), The A/UX Operating System (1 day), and The Macintosh User Interface for A/UX Programmers (4 days).

Conclusion

I know this resource kit isn't complete and I'm sure that I've left out somebody's favorite goodies, but this should give you some ideas about where to start. Have fun! The Macintosh can be as great a machine to work with as it is to use.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Tony Meadow is president and co-founder of Bear River Associates Inc., P.O. Box 1900, Berkeley, CA 94701. Bear River Associates is a software engineering department for hire. Tony has developed Macintosh applications since 1985. He is an active member of the Software Entrepreneurs' Forum. He and David Richey also develop and present technical courses about the Macintosh.

OVERVIEW:
MACINTOSH PROGRAMMER'S WORKSHOP

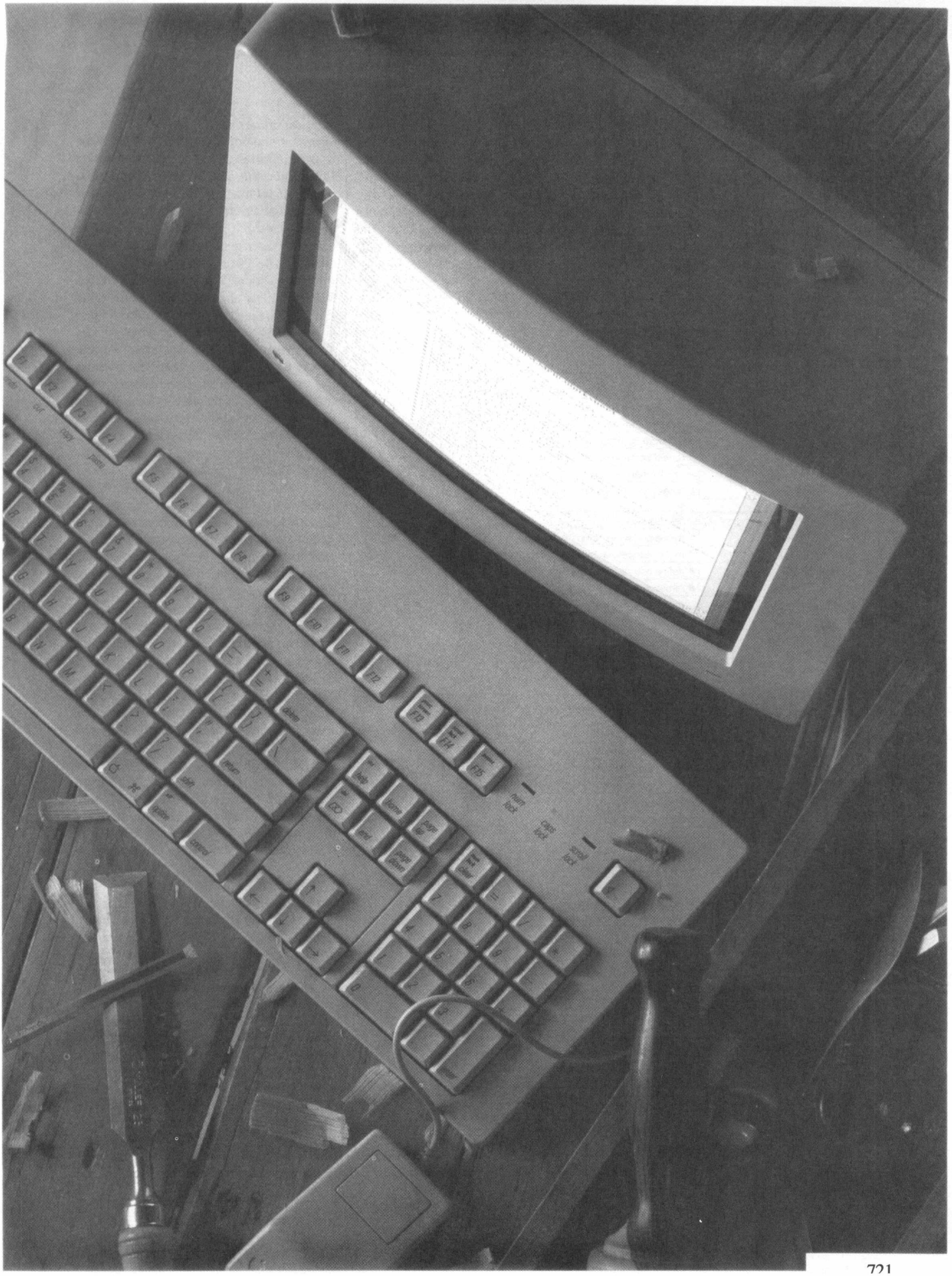
Apple's internal Macintosh development system is available to the rest of us.

by Dan Allen

As a professional environment for developing Macintosh software, the Macintosh Programmer's Workshop (MPW) is versatile. MPW was designed primarily for developing stand-alone, double-clickable applications, and many successful applications, including HyperCard, MacDraw, and More, have been developed with it. (See the sidebar, "Background, History, and Credits," for a glimpse of the people who have contributed to MPW.)

MPW supports integrated tools, which are generic line-oriented applications that run inside the MPW environment. Integrated tools include language translators, text tools, and other nongraphical tools. Unlike with standard applications, integrated tools can use the MPW shell environment and resources, which frees programmers from having to write a Macintosh application every time the need for a small utility arises.

Code is usually disguised as a resource of some kind in the Macintosh. Macintosh files contain two forks: a resource fork and a data fork. MPW is able to create many different resources, such as *INIT*s, *PACK*s, *MDEF*s, *WDEF*s, *DRVR*s, and desk accessories. In fact, the Macintosh ROMs themselves are built entirely with MPW. A new resource type, the HyperCard *XCMD*, allows HyperTalk to be extended by calling compiled resources. (For more information see "Introduction HyperCard Programming," page 56.



(continued from page 20)

The MPW Interface

MPW is a mixture of Smalltalk and the Unix system.

From Smalltalk, it inherits an integrated environment as well as the ability to interpret any text when users select it and press the Enter key. Because of effective use of RAM, the disk-based editor can quickly edit multimegabyte files. The editor is mainly mouse-based, but it also supports cursor keys and several keyboard shortcuts. The editor is built-in and other text editors do not work as effectively, which is annoying for those who have strong preferences about editors. The advantages of the integrated environment, however, outweigh the disadvantages of the built-in editor.

From Unix, MPW inherits the notion of a command shell. The command interpreter includes support for aliases, shell variables, structured constructs, I/O redirection, pipes, shell scripts, and the sublaunching of tools and applications from MPW. The command interpreter's history mechanism is simple and easy to use: Commands are maintained in the Worksheet, a window that is always open. The editor handles the Worksheet as it does any other open file. To execute a prior command, you simply scroll back to the start of the Worksheet, select the line (triple-clicking is a shortcut), and then press Enter or click with the mouse on the lower-left corner of the window. You can execute commands from any window, not just the Worksheet. MPW supports up to 12 open windows in addition to the Worksheet.

A window is a view into a file of the same name—a powerful concept when combined with I/O redirection. If the output of a tool is redirected to a currently open file, for example, the output will go to the window as well as to the file. It is easy to analyze tool output because one window contains various shell commands and a second, adjacent window contains the output from the commands.

MPW supports three types of commands: built-ins, tools, and scripts. If it receives a command that it does not recognize as a built-in, it searches a user-definable search path of directories, looking for a file of the same name. If the file is a regular MPW text file (a file of type *TEXT*), MPW interprets it as a shell-command script. If the file is an MPW tool (a file of type *MPST*), then MPW runs it as executable code.

Find-and-replace commands work on open windows, either in a literal mode similar to that found on most word processors or in a selection-expression mode in which you can issue complicated, cryptic commands (reminiscent of Unix's regular expressions) to do powerful text processing. Numeric and string expressions are evaluated in a similar way to the pattern portion of an *awk* script. You can specify regular expressions as you do in *grep*, and selections allow you to specify ranges of text in windows in powerful ways. Selection expressions are the most arcane aspect of MPW and have a steep learning curve, which is why they are underused by most programmers.

Structured Commands	
Begin	# group commands
Break	# break from For or Loop
Continue	# continue with next iteration of For or Loop
Evalute	# evaluate an expression
Execute	# execute command file in current scope
Exit	# exit from command file
For	# repeat commands once per parameter
If	# conditional command execution
Loop	# repeat commands until Break

Variable/Parameter Commands	
Ailas	# define and write command aliases
Export	# make variables available to commands
Set	# define or write Shell variables
Unalias	# remove aliases
Unexport	# remove variables available to commands
Unset	# remove Shell variable definitions
Parameters	# write parameters
Shift	# renumber command file positional parameters
Echo	# echo parameters
Quote	# echo quoted parameters

File System Commands	
Catenate	# concatenate files
Delete	# delete files and directories
Directory	# set or write the default directory
Duplicate	# duplicate files and directories
Eject	# eject volumes
Equal	# compare files and directories
Erase	# initialize volumes
Exists	# test existence of a file or directory
Files	# list files and directories
Mount	# mount volumes
Move	# move files and directories
Newer	# compare modification dates of files
NewFolder	# create folder
Rename	# rename files and directories
Setfile	# set file attributes
Unmount	# unmount volumes
Volumes	# list mounted volumes
Which	# determine what file the shell will execute

Misc. Commands	
Beep	# generate tones
Date	# write the date and time
Help	# write summary Information
Quit	# quit MPW
Shutdown	# shutdown/reboot the machine

Table 1: *Traditional Unix-system-like commands*

MPW inherits the ability to extend the system from both Smalltalk and Unix. From customizing a *UserStartup* shell script to designing custom menus and keyboard equivalents that are tied into shell commands all the way to writing new tools, MPW is designed to be configured the way the programmer wants it to be.

The Command Language

When you press the Enter key (or equivalent), MPW uses seven steps to interpret a command: alias substitution, evaluation of structured constructs, variable and command substitution, blank interpretation, file-name generation, I/O redirection, and execution.

Alias substitution allows you to define command names. Variables are referred to in braces. Command substitution, a powerful mechanism, occurs through the use of backquotes. File-name generation is available to all commands and is used to specify files through the use of selection expressions. I/O redirection allows you to send *stdin*, *stdout*, and *stderr* to arbitrary destinations, also specified by selection expressions. It is possible, for example, to run a tool on a selected portion of text in an open window.

The MPW shell's built-in command language consists of traditional Unix-like commands (see Table 1, page 22) and Mac-like commands (see Table 2, this page). The tables do not include the many command-line options that are applicable to each command.

The structured commands in Table 1 provide the basic constructs that support iteration and conditional branching in scripts. The variable/parameter commands allow scripts to pass information to and from each other in various ways. Scripts can call each other recursively and can support both local and global variables as well as multiple scoping levels.

The commands in Table 1 also provide a way to programmatically accomplish most of the operations that can be done in the Finder, such as copying and renaming files. For those with a Unix background, *Files* is very similar to *ls*, *Directory* to *cd*, and *NewFolder* to *mkdir*. MPW also includes a file of Unix aliases to make Unix users feel at home.

As Table 2 illustrates, MPW provides built-in scriptable commands for most of the actions that you would normally do with the mouse or keyboard, thus facilitating build scripts that automate the production of software.

MPW Tools

The advent of the MPW shell has produced MPW tools, a new class of applications that are distinct from normal applications or desk accessories. An MPW tool is similar to a standard Macintosh application, but it runs as part of the MPW shell and benefits from many services that the shell provides. An MPW tool is actually a coroutine that resides within the MPW shell's heap. The rules for writing tools are short. Tools do not need to initialize the various Mac Toolbox managers or deal with menus or events because these services are performed for them by the MPW shell.

How are tools launched and how does an MPW tool fit into the Macintosh architecture? A major portion of the shell continues to be active and resident in memory during the execution of a tool, so the first step is to unload any code segments that are not needed during the execution of a tool in order to allow it more available heap space.

The second step is to open the tool's resource fork and allocate a cache entry. Tools are cached in memory when they are first executed, and therefore subse-

Window Commands	
New	# open new file in window
Open	# open file in window
Target	# make window the target window
Close	# close a window
Save	# save contents of window
Revert	# revert to saved document
MoveWindow	# move window to x,y
SizeWindow	# make window be x by y
StackWindows	# arrange windows stacked
TileWindows	# arrange windows to be tiled
ZoomWindow	# zoom target window to full size
Windows	# list open windows

Menu Commands	
Add Menu	# add user-defined menu item
Delete Menu	# delete user-defined menus and items

Dialog Commands	
Alert	# display alert box
Confirm	# display confirmation dialog
Request	# request text from a dialog

Text Commands	
Find	# find and select a text pattern
Replace	# replace the selection
Mark	# set a marker in a window
Unmark	# delete a marker in a window
Tab	# set tab setting of a window
Font	# set font setting of a window
Adjust	# adjust lines
Align	# align text to left margin

Clipboard Commands	
Undo	# undo last edit in target window
Cut	# copy selection to Clipboard and delete it
Copy	# copy selection to Clipboard
Paste	# replace selection with Clipboard contents
Clear	# clear the selection

Table 2: *Macintosh-like commands*

(continued from page 23)
quent executions of a cached tool are much quicker. This speed advantage is noticeable when correcting a compilation error and recompiling code. Up to ten tools can be cached in RAM, with the oldest tools being purged from memory as additional space is required.

The third step in the launching of an MPW tool is to create a separate and distinct A5 World for the tool. (An A5 World consists of areas of memory that depend upon the value contained in the *A5* register of the 68xxx micro processor. Such areas include the application's globals, QuickDraw's globals, and the intersegment jump table.) What this means is that the MPW shell does a type of context switch by setting up an area for the tool in the shell's own application heap that contains the tool's own A5 World. The MPW shell takes care of allocating a nonrelocatable block for the globals and also sets up the jump table (code segment 0 of any Mac application or tool). The tool's stack area, however, is shared with the stack of the MPW shell, which reduces memory requirements.

The fourth step is to set up the environment area, which is an area in memory containing the parameters being passed to the tool from the shell. These are accessed in C, for example, via the standard *argv/argc* convention.

Finally, the shell does a quick check of the heap for consistency and then calls the first routine in the tool's jump table. The MPW tool then effectively becomes the application in control.

Many of the common routines that an MPW tool may call, such as the file system and memory allocation routines, are patched out or intercepted by the MPW shell, thus allowing it to do I/O redirection and perform its windows-over-files abstraction. When these routines are called by a tool, or when a *readln* or *printf* instruction is called, the flow of execution returns to the shell while it handles the tool's request. Typical tools contain many instances in which the path of execution is transferred back and forth (transparently to the programmer) between the MPW shell and the MPW tool.

After the tool terminates, the shell automatically performs several cleanup operations. It retrieves the status from the environment area, closes any files left open by the tool, and then frees up any memory that was allocated by the tool. These operations are possible because the shell intercepts the memory allocation and file system calls. Future tools are thus given a clean environment in which to run. If a tool goes into an infinite loop, or if you want to terminate the execution of a tool, the shell provides a periodic vertical blanking (VBL) task that checks for a command period keyboard sequence, which will force a tool to be aborted and the environment to be cleaned up.

The real utility of an MPW tool is that generic code written for more traditional tty environments runs in the Macintosh environment without any additional code support. *readln*, *writeln*, *scanf*, and *printf* all work without the programmer's having to write a set of special QuickDraw commands to support them. Most utilities written for the Unix environment are especially good candidates for an easy port to MPW tools.

Asm	# 68020 macro assembler
Backup	#generates list of dup cmds based on file dates
C	# 68020 C compiler
Canon	# spell checks indentifiers based on canonical list
Commando	# dialog interface to MPW tools
Compare	# compares two text files, prints list of differences
Count	# counts characters and lines
DeRez	# de-compiles resourses to Rez format
DumpCode	# disassembles resources
DumpObj	# disassembles object code modules
Entab	# converts between spaces & tabs
FileDiv	# splits up large text files into many small files
GetErrorText	# display error msgs based on msg number
GetFileName	# display a Standard File dialog box
GetListItem	# display items for selection in a dialog box
Lib	# object code librarian
Link	# links object files: strips dead code
MacsBug	# assembly level debugger/disassembler
Make	# generates build commands based on file dates
MakeErrorFile	# creates error message textfile
Pascal	# 68xxx Pascal & Object Pascal compiler
PasMat	# text pretty-printer for Pascal sources
PasRef	# generates full xref listing for Pascal sources
PerformReport	# generates a performance report
Print	# prints files to LaserWriter & ImageWriter
ProcNames	# displays Pascal procedure and function names
ResEdit	# edits resources interactively (application)
ResEqual	# compares the resources in two files
Rez	# compiles resources from text description
RezDet	# lists resources: detects bad resources
Search	# searches multiple files for text pattern
SetVersion	# maintains version and revision number
Translate	# translates characters

Table 3: *Tools for the Macintosh Programmer's Workshop*

Standard MPW Tools

MPW includes many *standard tools. It* supports several languages, including assembly language, C, Pascal, and a special resource-oriented language called Rez. Various text tools, such as search, count, canon, compare, backup, entab, and translate help maintain source files. It also has a variety of linker- oriented tools, including a librarian, and various disassembler tools for examining generated object code. Also available is a performance package that makes profiling code possible. Table 3, this page, lists the tools that make up the MPW system.[1]

MPW Assembly Language

The premier language of MPW is Asm, written by Ira Ruben. Asm generates code for the entire 68xxx line of processors, including the 68000, 68010, 68020, 68030, 68851, 68881, and 68882 processors. MPW also includes a full set of equates files that support the Toolbox (MPW 2.0 also supports the Mac SE and Mac II) as well as sample programs in assembly language for an application, a desk accessory, and a tool. One entire volume of the documen-

Listing One

```
/* Benchmark by Dan Allen 9/18/87, Apple Computer */
/*
*
*           NAME
*                       sieve - prime number benchmark
*
*           SYNOPSIS
*                       sieve
*
*           DESCRIPTION
*                       sieve generates prime numbers via the method after Erathosthenes.
*                       It is the standard Byte magazine benchmark.
*
*           AUTHOR
*                       (a)             Tim Field, Byte Magazine, November 1985, p.275f.
*                       (B)             Dan Allen & Apple Computer, Inc. 1985,1986.
*
*/

#include <stdio.h>

pascal long TickCount() extern 0xA975;
pascal long Debugger() extern 0xA9FF;

#define MAXITR              100

#define TRUE                1
#define FALSE               0
#define SIZE                8190

static char flags[SIZE+1];

main()
{
            int                             i,prime,k,count,iter;
            long            time;

            printf("Generating primes_\n"); fflush(stdout);
            time = TickCount();
            for (iter = 1; iter <= MAXITR; iter++) {
                        count = 0;
                        for (i=0; i <= SIZE; i++)
                                    flags[i] = TRUE;
                        for (i=0; i <= SIZE; i++) {
                                    if (flags[i]) {
                                        prime = i + i + 3;
                                        for (k = i + prime; k <= SIZE; k += prime)
                                                    flags[k] = FALSE;
                                        count ++;
                                    }
                        }
            }
            time = TickCount() - time;
            printf("Time = %.3f seconds\n", time/60.0);
            printf("%u iterations\n", MAXITR);
            printf("%u primes generated\n", count);
            return(0);
}
```

End Listing One

Listing Two

```
/* Benchmark by Dan Allen 10/2/1986, Apple Computer */

/*
 *      The following program contains statements of a high-level programming
 *      language (C) in a distribution considered representative:
 *
 *      assignments                     53%
 *      control statements              32%
 *      procedure, function calls       15%
 *
 *      100 statements are dynamically executed.  The program is balanced with
 *      respect to the three aspects:
 *              - statement type
 *              - operand type (for simple data types)
 *              - operand access
 *                      operand global, local, parameter, or constant.
 *
 *      The combination of these three aspects is balanced only approximately.
 *
```

(continued on next page)

Listing Two *(Listing continued)*

```
 *      The program does not compute anything meaningfull, but it is
 *      syntactically and semantically correct.
 *
 */

/* Accuracy of timings and human fatigue controlled by next two lines */
#define LOOPS   50000           /* Use this for slow or 16 bit machines */
/*#define LOOPS 500000          /* Use this for faster machines */

/* Compiler dependent options */
#undef  NOENUM                  /* Define if compiler has no enum's */
#undef  NOSTRUCTASSIGN          /* Define if compiler can't assign structures */
/* define only one of the next two defines */
/*#define TIMES                   /* Use times(2) time function */
#define TIME                  /* Use time(2) time function */

/* define the granularity of your times(2) function (when used) */
#define HZ      60              /* times(2) returns 1/60 second (most) */
/*#define HZ    100             /* times(2) returns 1/100 second (WECo) */

/* for compatibility with goofed up version */
/* #define GOOF  Define if you want the goofed up version */

#ifdef GOOF
char    Version[] = "1.0";
#else
char    Version[] = "1.1";
#endif

#ifdef  NOSTRUCTASSIGN
#define structassign(d, s)      memcpy(&(d), &(s), sizeof(d))
#else
#define structassign(d, s)      d = s
#endif

#ifdef  NOENUM
#define Ident1  1
#define Ident2  2
#define Ident3  3
#define Ident4  4
#define Ident5  5
typedef int     Enumeration;
#else
typedef enum    {Ident1, Ident2, Ident3, Ident4, Ident5} Enumeration;
#endif

typedef int     OneToThirty;
typedef int     OneToFifty;
typedef char    CapitalLetter;
typedef char    String30[31];
typedef int     Array1Dim[51];
typedef int     Array2Dim[51][51];

struct  Record
{
        struct Record           *PtrComp;
        Enumeration             Discr;
        Enumeration             EnumComp;
        OneToFifty              IntComp;
        String30                StringComp;
};

typedef struct Record   RecordType;
typedef RecordType *    RecordPtr;
typedef int             boolean;

#define NULL            0
#define TRUE            1
```



```
#define FALSE           0

#ifndef REG
#define REG
#endif

extern Enumeration      Func1();
extern boolean          Func2();

#ifdef TIMES
#include <sys/types.h>
#include <sys/times.h>
#endif

pascal long TickCount() extern 0xA975;
long time()
{
            return(TickCount()/60);
```

(continued from page 26)

tation that comes with MPW is devoted to the many features and options Asm contains.[2] Asm is fast: It assembles instructions at rates greater than 40,000 lines per minute on a Mac II.[3]

A highlight of Asm is its powerful macro processor, written by Fred Forsman. Using macros provides for a higher level of abstraction than is normally available when using assembly language. Included with MPW are a set of structured macros (written by Ira Ruben) that implement many of the structured constructs that are available to C and Pascal programmers. When using the structured macros, assembly-language code looks similar to Pascal and yet retains the efficiency of assembly language. Asm also supports the use of interactive assembly, by which lines of assembly-language code can be typed in and assembled on the fly. This is made possible by the tight coupling between Asm and the shell. (Any tool that supports *stdin* can do this.)

Another interesting use of Asm's macro processor is illustrated by a set of macros that implement object assembly language. These object macros were written by Ken Doyle, who also wrote the object extensions to MPW Pascal. Object assembly language is 68xxx assembly-language code that is interlanguage callable with Object Pascal. These object macros are also included with MPW and allow time-critical portions of object-oriented programs to be recoded in assembly language for greater speed.

MPW C

MPW C is a version of the Green Hills C compiler designed specially for Apple and is similar to the version that was available for the Lisa Workshop. It has the usual obligatory post-K&R extensions to C (such as *enum*) that are described well in Harbison and Steele.[4] MPW C is similar to C compilers found on VAXs in that it uses 32-bit integers, and although some consider it an inefficiency, this feature makes porting C code from the Unix world effortless. MPW C is particularly good in its global register allocation strategy that results in above-average generated code. Version 2.0 of MPW C introduced support for generating 68020 and 68881 code.

MPW C is not without its quirks, however. It has a proclivity for allocating global data—a problem for those

writing special types of code such as drivers and desk accessories. Using a literal string or even a floating-point literal causes global data to be allocated! Its compilation time is slow, and occasionally its generated code is unneeded (not inaccurate, just wasted). The compiler is big (233K), and its generated code, although fast, can also be big, especially when the C libraries are added.

The manual for MPW C contains a good delta description of the language as it varies from K&R or H&S. The manual does not include a full language reference, although it does have pages for the C library routines. It also includes a condensed quick-reference listing of the *Inside Macintosh* calls. The ideal solution is to use three manuals that complement each other: H&S, *The C Programmer's Handbook*,[5] and the supplied *Macintosh Programmer's Workshop C Reference*.[6]

MPW Pascal

MPW Pascal is a descendant of Lisa Pascal and conforms closely to the ANSI standard for Pascal. Its enhancements to standard Pascal are major and *significant and* require a bit of explanation.

Silicon Valley Software (SVS) originally wrote Lisa Pascal for Apple in 1981, although Apple has maintained it for years now. It supported Units, a method of separate compilation that provides an Interface as well as an Implementation section for each module of code, thus providing similar facilities to Modula-2.[7]

An early version of object-oriented programming was supported as the language further evolved into a language called Clascal. Clascal began in 1983, when Larry Tesler (formerly of Xerox PARC) asked Chris Franklin to implement classes. It was later enhanced by Al Hoffman and then by Ira Ruben. Early in 1985, a team including Larry Tesler and Nikolaus Wirth created Object Pascal, a superset of Pascal and the successor to Clascal. Object Pascal also supports the concepts of objects, classes, and inheritance but in a simpler and clearer way than does Clascal. Ken Doyle finished things up by writing the Object Pascal extensions to the MPW Pascal compiler. MPW Pascal is thus a full Object Pascal compiler.

In addition to Units with their facility for separate compilation, MPW Pascal

Listing Two *(Listing continued)*

```
}

main()
{
    Proc0();
    exit(0);
}

/*
 * Package 1
 */
int             IntGlob;
boolean         BoolGlob;
char            Char1Glob;
char            Char2Glob;
Array1Dim       Array1Glob;
Array2Dim       Array2Glob;
RecordPtr       PtrGlb;
RecordPtr       PtrGlbNext;

Proc0()
{
        OneToFifty              IntLoc1;
        REG OneToFifty          IntLoc2;
        OneToFifty              IntLoc3;
        REG char                CharLoc;
        REG char                CharIndex;
        Enumeration             EnumLoc;
        String30                String1Loc;
        String30                String2Loc;
        extern char             *malloc();

#ifdef TIME
        long                    time();
        long                    starttime;
        long                    benchtime;
        long                    nulltime;
        register unsigned int   i;

        starttime = time( (long *) 0);
        for (i = 0; i < LOOPS; ++i);
        nulltime = time( (long *) 0) - starttime; /* Computes o'head of loop */
#endif
#ifdef TIMES
        time_t                  starttime;
        time_t                  benchtime;
        time_t                  nulltime;
        struct tms              tms;
        register unsigned int   i;

        times(&tms); starttime = tms.tms_utime;
        for (i = 0; i < LOOPS; ++i);
        times(&tms);
        nulltime = tms.tms_utime - starttime; /* Computes overhead of looping */
#endif

        PtrGlbNext = (RecordPtr) malloc(sizeof(RecordType));
        PtrGlb = (RecordPtr) malloc(sizeof(RecordType));
        PtrGlb->PtrComp = PtrGlbNext;
        PtrGlb->Discr = Ident1;
        PtrGlb->EnumComp = Ident3;
        PtrGlb->IntComp = 40;
        strcpy(PtrGlb->StringComp, "DHRYSTONE PROGRAM, SOME STRING");
#ifndef GOOF
        strcpy(String1Loc, "DHRYSTONE PROGRAM, 1'ST STRING");   /*GOOF*/
#endif
        Array2Glob[8][7] = 10;  /* Was missing in published program */

/*****************
-- Start Timer --

DHRYSTON.TXT     05/19/88     5:38 PM     Galley 3 of 5

*****************/
#ifdef TIME
        starttime = time( (long *) 0);
#endif
#ifdef TIMES
        times(&tms); starttime = tms.tms_utime;
#endif
        for (i = 0; i < LOOPS; ++i)
        {

                Proc5();
                Proc4();
                IntLoc1 = 2;
                IntLoc2 = 3;
```

(Listing continued on next page)

Listing Two *(Listing continued)*

```
                strcpy(String2Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
                EnumLoc = Ident2;
                BoolGlob = ! Func2(String1Loc, String2Loc);
                while (IntLoc1 < IntLoc2)
                {
                        IntLoc3 = 5 * IntLoc1 - IntLoc2;
                        Proc7(IntLoc1, IntLoc2, &IntLoc3);
                        ++IntLoc1;
                }
                Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
                Proc1(PtrGlb);
                for (CharIndex = 'A'; CharIndex <= Char2Glob; ++CharIndex)
                        if (EnumLoc == Func1(CharIndex, 'C'))
                                Proc6(Ident1, &EnumLoc);
                IntLoc3 = IntLoc2 * IntLoc1;
                IntLoc2 = IntLoc3 / IntLoc1;
                IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1;
                Proc2(&IntLoc1);
        }

/*****************
-- Stop Timer --
*****************/

#ifdef TIME
        benchtime = time( (long *) 0) - starttime - nulltime;
        printf("Dhrystone(%s) time for %ld passes = %ld\n",
                Version,
                (long) LOOPS, benchtime);
        printf("This machine benchmarks at %ld dhrystones/second\n",
                ((long) LOOPS) / benchtime);
#endif
#ifdef TIMES
        times(&tms);
        benchtime = tms.tms_utime - starttime - nulltime;
        printf("Dhrystone(%s) time for %ld passes = %ld\n",
                Version,
                (long) LOOPS, benchtime/HZ);
        printf("This machine benchmarks at %ld dhrystones/second\n",
                ((long) LOOPS) * HZ / benchtime);
#endif

}

Proc1(PtrParIn)
REG RecordPtr   PtrParIn;
{
#define NextRecord      (*(PtrParIn->PtrComp))

        structassign(NextRecord, *PtrGlb);
        PtrParIn->IntComp = 5;
        NextRecord.IntComp = PtrParIn->IntComp;
        NextRecord.PtrComp = PtrParIn->PtrComp;
        Proc3(NextRecord.PtrComp);
        if (NextRecord.Discr == Ident1)
        {
                NextRecord.IntComp = 6;
                Proc6(PtrParIn->EnumComp, &NextRecord.EnumComp);
                NextRecord.PtrComp = PtrGlb->PtrComp;
                Proc7(NextRecord.IntComp, 10, &NextRecord.IntComp);
        }
        else
                structassign(*PtrParIn, NextRecord);

#undef  NextRecord
}

Proc2(IntParIO)
OneToFifty      *IntParIO;
{
        REG OneToFifty          IntLoc;
        REG Enumeration         EnumLoc;

        IntLoc = *IntParIO + 10;
DHRYSTON.TXT     05/19/88     5:38 PM     Galley 4 of 5

        for(;;)
        {
                if (Char1Glob == 'A')
                {
                        --IntLoc;
                        *IntParIO = IntLoc - IntGlob;
                        EnumLoc = Ident1;
                }
                if (EnumLoc == Ident1)
                        break;
        }
}
```

(continued from page 29)

has also been extended to support conditional compilation and compile-time variables, bit operators, short-circuit Boolean operators, *Leave* and *Cycle* statements (similar to *break* and *continue* in C), type coercion, and many other concepts. MPW Pascal overcomes most of Brian Kernighan's objections to Pascal included in his famous memo "Why Pascal Is Not My Favorite Programming Language."[8]

New in the MPW 2.0 Pascal compiler was the facility to generate 68020 and 68881 code. Support for large arrays was also added by dynamically allocating them on the heap. These additions combine to make MPW Pascal a good choice for general scientific and numeric programming, as MPW Pascal fully supports SANE (Standard Apple Numerics Environment). MPW Pascal comes with its own language reference manual detailing all aspects of this extended language.[9]

MPW Linker Tools

MPW has been designed from the onset to provide a multilingual environment. It is possible to link the object code files that result from applications written in the three standard languages—assembly language, C, and Pascal. The MPW system itself is built using all three of these languages. Other languages are now becoming available for MPW, and their object code can also be linked in.[10]

The Link tool can automatically strip unused code but at a cost: Linking HyperCard on a Mac II, for example, takes about 42 seconds; the MPW shell takes 35 seconds.[11] One reason why the linker is so slow is simply that a large number of symbols are being pushed through it. Nevertheless, because MPW provides interlanguage linking support, solutions to performance problems in an application can be solved with Bill Atkinson's success formula: 95 percent Pascal and 5 percent assembly language.[12]

MPW Resource Tools

rez is a language that is similar in its syntax to C but is especially crafted for describing Macintosh resources, such as *ALRT*s, *DLOG*s, *DITL*s, and *MENU*s. These and many other resources can be described textually in the *rez* language. User-defined types are easily constructed and aid in making programs easily localizable to foreign languages. As an aid to using *rez*, MPW provides a compan-

ion resource decompiler called *derez* that can derive source from existing resources.

Other than the MPW shell itself, *resedit*—a familiar utility program to most Mac programmers, not to mention many power users—is the only other application in MPW. *resedit* is an interactive tool for creating, modifying, deleting, and moving resources. It was originally written by Steve Capps, who incidentally wrote many versions of the Finder. Rony Sebok wrote the template editor, and Gene Pope wrote most of the other pickers and editors. Using MPW Pascal, you can write custom editors and add them to the many editors already present in *resedit*. Sample code to extend *resedit* is included with MPW Pascal.

A common way to work with resources is to create resources with *resedit*, use *derez* to decompile them, and then maintain and tweak the *rez* sources.

Two other resource tools are also provided with MPW. *resequal* is a comparison tool, similar to *diff* in Unix. *resequal*, however, compares resources rather than text, showing all differences between two files. *rezdet* is a resource detective that verifies a resource fork and can also list the contents of a resource fork in several different formats.

Object-Oriented Programming

The MPW assembler with its object macros and the MPW Pascal compiler both support object-oriented programming. To a first approximation, the following equation defines object-orientedness:[13]

object-oriented =
objects + classes + inheritance

Objects are the atomic entities of object-oriented programming. Objects consist of a data structure and its related methods (procedures) that can manipulate objects. Objects are specific instances of a class. Classes are arranged hierarchically, with descendant objects referred to as subclasses and ancestor objects referred to as superclasses. Objects of a subclass inherit properties from their ancestor objects. Objects can send, receive, and respond to messages sent by other objects.[14]

Data abstraction is an orthogonal language attribute from object-orientedness. An abstract data type as supported in Ada or C++ is a data structure and a set of associated operations that are the

Listing Two *(Listing continued)*

```
Proc3(PtrParOut)
RecordPtr       *PtrParOut;
{
        if (PtrGlb != NULL)
                *PtrParOut = PtrGlb->PtrComp;
        else
                IntGlob = 100;
        Proc7(10, IntGlob, &PtrGlb->IntComp);
}

Proc4()
{
        REG boolean     BoolLoc;

        BoolLoc = Char1Glob == 'A';
        BoolLoc |= BoolGlob;
        Char2Glob = 'B';
}

Proc5()
{
        Char1Glob = 'A';
        BoolGlob = FALSE;
}

extern boolean Func3();

Proc6(EnumParIn, EnumParOut)
REG Enumeration EnumParIn;
REG Enumeration *EnumParOut;
{
        *EnumParOut = EnumParIn;
        if (! Func3(EnumParIn) )
                *EnumParOut = Ident4;
        switch (EnumParIn)
        {
        case Ident1:    *EnumParOut = Ident1; break;
        case Ident2:    if (IntGlob > 100) *EnumParOut = Ident1;
                        else *EnumParOut = Ident4;
                        break;
        case Ident3:    *EnumParOut = Ident2; break;
        case Ident4:    break;
        case Ident5:    *EnumParOut = Ident3;
        }
}

Proc7(IntParI1, IntParI2, IntParOut)
OneToFifty      IntParI1;
OneToFifty      IntParI2;
OneToFifty      *IntParOut;
{
        REG OneToFifty  IntLoc;

        IntLoc = IntParI1 + 2;
        *IntParOut = IntParI2 + IntLoc;
}

Proc8(Array1Par, Array2Par, IntParI1, IntParI2)
Array1Dim       Array1Par;
Array2Dim       Array2Par;
OneToFifty      IntParI1;
OneToFifty      IntParI2;
{
        REG OneToFifty  IntLoc;
        REG OneToFifty  IntIndex;

        IntLoc = IntParI1 + 5;
        Array1Par[IntLoc] = IntParI2;
        Array1Par[IntLoc+1] = Array1Par[IntLoc];
        Array1Par[IntLoc+30] = IntLoc;
        for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)
                Array2Par[IntLoc][IntIndex] = IntLoc;
        ++Array2Par[IntLoc][IntLoc-1];
        Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];
        IntGlob = 5;
DHRYSTON.TXT     05/19/88     5:38 PM     Galley 5 of 5

}

Enumeration Func1(CharPar1, CharPar2)
CapitalLetter   CharPar1;
CapitalLetter   CharPar2;
{
        REG CapitalLetter       CharLoc1;
        REG CapitalLetter       CharLoc2;
```

(continued on next page)

Listing Two *(Listing continued)*

```
        CharLoc1 = CharPar1;
        CharLoc2 = CharLoc1;
        if (CharLoc2 != CharPar2)
                return (Ident1);
        else
                return (Ident2);
}

boolean Func2(StrParI1, StrParI2)
String30        StrParI1;
String30        StrParI2;
{
        REG OneToThirty         IntLoc;
        REG CapitalLetter       CharLoc;

        IntLoc = 1;
        while (IntLoc <= 1)
                if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
                {
                        CharLoc = 'A';
                        ++IntLoc;
                }
        if (CharLoc >= 'W' && CharLoc <= 'Z')
                IntLoc = 7;
        if (CharLoc == 'X')
                return(TRUE);
        else
        {
                if (strcmp(StrParI1, StrParI2) > 0)
                {
                        IntLoc += 7;
                        return (TRUE);
                }
                else
                        return (FALSE);
        }
}

boolean Func3(EnumParIn)
REG Enumeration EnumParIn;
{
        REG Enumeration EnumLoc;

        EnumLoc = EnumParIn;
        if (EnumLoc == Ident3) return (TRUE);
        return (FALSE);
}

#ifdef  NOSTRUCTASSIGN
memcpy(d, s, l)
register char   *d;
register char   *s;
register int    l;
{
        while (l--) *d++ = *s++;
}
#endif
```

End Listing Two

Listing Three

```
/* Square Root Benchmark by Dan Allen 2/26/88 */

#include <stdio.h>
#include <math.h>

#define     NUM             100000

pascal long TickCount() extern 0xA975;

main(argc, argv)
            int             argc;
            char            *argv[];
{
            long                                        i, ticks;
            extended                        sum = 0.0;

            printf("Adding first %u square roots_\n",NUM);
            fflush(stdout);
            ticks = TickCount();

            for (i = 1; i <= NUM; i++)
                        sum += sqrt( (extended) i);

            ticks = TickCount() - ticks;
            printf("Time = %.2f seconds\n",ticks/60.0);
            printf("Sum  = %.18f\n",sum);
            return(0);
}
```

End Listing Three

(continued from page 31)

only way to access the private data structure. An object-oriented language such as Object Pascal, however, can access or modify any field of an object directly, as if the object were a record. C++ supports object-oriented programming by allowing members of an object to be public or private. Ada does not support inheritance, which means that Ada is not an object-oriented language.

MacApp

MacApp (Macintosh Application) is another part of MPW that originally developed out of the Lisa program from a project called the Lisa Toolkit. The Lisa Toolkit was developed by Larry Tesler, Larry Rosenstein, and Pete Young and was written using Clascal. MacApp was a completely new system designed by Larry Rosenstein, Larry Tesler, Scott Wallace, and Ken Doyle and implemented by Larry Rosenstein and Scott Wallace. Early versions ran in the Lisa Workshop, but the official 1.0 release and all subsequent releases have been for MPW.

MacApp is a separately sold product built upon Object Pascal, so use of MacApp requires MPW and MPW Pascal. Essentially, it is a huge library of routines consisting of some 29,000+ lines of Object Pascal and 1,600+ lines of object assembly language.

MacApp fully implements the standard Macintosh user interface in a generic Macintosh application that the programmer builds upon, thus greatly reducing the amount of code that needs to be written without reducing the ensuing quality. A programmer would, for example, need only to write routines to draw items in windows or to write files to the disk. The MacApp libraries support desk accessories, menus, multiple documents, error handling, the Clipboard, and printing as well as scrolling, moving, resizing, and zooming windows.

With the help of MacApp, you can build significant Macintosh applications in a few weeks rather than in several months. What is even more important is that they comply with the recommended guidelines for developing applications and therefore are more robust to changes in system software. Users also benefit from applications that are consistent with the Mac interface and hence are easy to use.

MacApp includes the full source code

for the MacApp libraries as well as many sample programs illustrating object-oriented Macintosh programming. Also included is a cookbook of tips and routines useful in the creation of Macintosh applications.

A Sample MPW Shell Script

The sample script in Example 1, this page, canonizes the text files that ship with MPW to have the same font, font size, tab settings, and window size. It is an example of a script that automates an otherwise laborious process.

The pound sign *(#)* serves as a comment token, with comments being valid until the next carriage return. The first line of the script creates and sets a local shell variable named *today* to hold the current date and time. Command substitution takes place by use of backquotes: the output of the *date* command is substituted for what is between the backquotes. Double quotes group the two resultant strings.

The second line of the script changes *directory* into the standard MPW directory, looking up that location in a shell variable of the same name. Braces are used for referring to the contents of shell variables and usually require quoting if the variable contains spaces or other special characters.

The third line of the script begins a *for* loop. *for* loops iterate through a provided list of items—in this case a list of directories found in the main MPW folder. This list is also dynamically generated by using backquotes and command substitution. The colon denotes the current directory.

The body of the loop changes directory into a folder and then uses another backquoted expression, this time to list the files of type *TEXT* that exist within

Listing Four

```
/* Hilbert Matrix Benchmark by Dan Allen 2/26/88 */

#include <stdio.h>
#include <math.h>

pascal long TickCount() extern 0xA975;
pascal long Debugger() extern 0xA9FF;

#define MAXARRAY            26
#define NUM                                 100

typedef extended matrix[MAXARRAY][MAXARRAY];
typedef extended vector[MAXARRAY];

main(argc,argv)
            int                             argc;
            char            *argv[];
{
            matrix                          a;
            vector                          b,x;
            extended        det,sum,temp;
            int                                             i,j,k,upper_bound,ticks;
            void                            system(), printVector(), printError();

            upper_bound = (argc == 2) ? atoi(argv[1]) : 10;
            if (upper_bound >= MAXARRAY) {
                        fprintf(stderr,"### Array size out of bounds\n");
                        return(1);
            }

            ticks = TickCount();
            for (k = 1; k <= NUM; k++) {
                        for (i = 1; i <= upper_bound; i++) {
                                    sum = 0.0;
                                    for (j =1; j <= upper_bound; j++) {
                                                temp = 1.0/(i+j-1);
                                                a[i][j] = temp;
                                                sum += temp;
                                    }
                                    b[i] = sum;
                        }
                        system(upper_bound,a,b,x,&det);
            }
            ticks = TickCount() - ticks;

            printVector(upper_bound,x);
            printError(upper_bound,x);
            printf("Det  = %.18e\n",det);
            printf("Time = %.2f seconds for %u iterations.\n",ticks/60.0,NUM);
            return(0);
}


static void system(n,a,b,x,det)
            int                                             n;
            matrix                          a;
            vector                          b,x;
            extended        *det;
{
            int                                             i,j,m;
            extended        deter, k;
```

(continued on next page)

```
# StdMPW Canonizes all MPW Textfiles
# By Dan Allen 7/13/87 set today "`date -s -d` 12:00pm"
directory "{mpw}"
for i in : `files -d`
   directory "{i}"
     for j in `files -t TEXT`
            setfile -a l "{j}"
            target "{j}"
                  font Monaco 9 ; tab 4 ; find
                  movewindow 36 22 ; sizewindow 473 280
                  close
     end
     if "{i}" != ":"
            directory ::
     end
end
```

Example 1: *A sample script to canonize text files*

Listing Four *(Listing continued)*

```
            deter = 1;
            for ( m = 1; m <= n-1; m++) {
                        deter = a[m][m];
                        for (i=m+1; i<=n; i++) {
                                    k = a[i][m]/a[m][m];
                                    for (j = m+1; j <= n; j++)
                                                a[i][j] -= k*a[m][j];
                                    b[i] -= k*b[m];
                        }
            }
            deter *= a[n][n];
            for (m = n; m >= 1; m--) {
                        x[m] = b[m]/a[m][m];
                        for (i = 1; i <= m-1; i++)
                                    b[i] -= x[m]*a[i][m];
            }
            *det = deter;
}


static void printError(n,a)
            int                         n;
            vector          a;
{
            int                                                 i;

            extended        totalErr = 0.0;

            for (i = 1; i <= n; i++)
                        totalErr += fabs(a[i]-1.0);
            printf("Err  = %.18e\n",totalErr);
}


static void printVector(n,a)
            int                         n;
            vector          a;
{
            int                         i;

            for (i = 1; i <= n; i++)
                        printf("%d  %.18f\n",i,a[i]);
}


static void printMatrix(n,a)
            int                         n;
            matrix          a;
{
            int                         i,j;

            for (i = 1; i <= n; i++) {
                        for (j = 1; j <= n; j++)
                                    printf("%.6f   ",a[i][j]);
                        printf("\n");
            }
}
```

End Listing Four

Listing Five

```
/* Trig Accuracy Benchmark by Dan Allen 2/26/88 */

#include <stdio.h>
#include <math.h>

#define     NUM             100

pascal long TickCount() extern 0xA975;

main(argc, argv)
            int             argc;
            char            *argv[];
{
            long                                                ticks, count;
            extended                        i, x, y, sum;
```

(continued from page 33)

the directory. If the file has been locked, it is unlocked using the *setfile* command and then opened as the target next-to-top window with the *target* command. The target window denotes not the topmost window but the next-to-topmost window. If commands do not specify a window, then the target window is the automatic default receiver of commands.

After the window has been opened as the target window, the *font* and *tab* commands are used to change these settings. Multiple commands are placed on the same line by using the semicolon as a separator. The *find* command then uses the bullet selection expression to specify placing the cursor at the start of the file. The *movewindow* command places the upper-left corner of the window at the coordinates (36,22) (pixels are measured from the bottom-left corner of the menu bar). The *sizewindow* command changes the windows size to 473 pixels wide by 280 pixels high. The window is then closed, with these settings automatically being saved. (If the contents of the file had been changed, then you would have been prompted with a dialog. It is possible to override the dialog by using an option to the *close* command, but the changes in this example are only of a cosmetic nature and are thus always saved to the resource fork of the file.)

The end of the script illustrates the conditional *if* statement and the shell's use of C-like expressions. If the current directory is not at the main MPW level, the colon-colon argument to the *directory* command simply pops back up a level in the HFS hierarchy.

Benchmarks

I tested MPW with a suite of five benchmarks—Sieve, Dhrystone, Sqrt, Hilbert, and Trig. The hardware platform was a Macintosh II (with its standard 16-MHz 68020 and 16-MHz 68881) with 5-Mbyte RAM and an Apple HD-80SC internal hard disk. MPW 2.0 ran in a 1,024K partition under MultiFinder 1.0.

Unless otherwise specified, MPW C used only the *–g* option to generate MacsBug symbols. MPW Pascal defaults to generating MacsBug symbols. I also used the *–r* option to suppress range checking. The hardware floating-point versions of the benchmarks for both MPW C and MPW Pascal also used the *–mc68020*, *–mc68881*, *–elems881*, and *–x149* options for the

greatest possible speed.

The build time was measured by bracketing the build using the built-in *date* command; its display precision is good only to a second. The actual execution time was timed using calls to the *TickCount* routine built into the Macintosh Toolbox. This function is good to 1/60th of a second. Object code size was determined in separate compilations of the code using the *–p* (progress) option found in the compilers. And finally, the executable code size was determined by the *files* command.

To test integer math, I used the ubiquitous Sieve benchmark (see Listing One, page 27) of *Byte* magazine fame. It

```
    printf("Computing trig accuracy 0 to 3 step .01 %d times_\n",NUM);
    fflush(stdout);
    ticks = TickCount();

    for (count = 1; count <= NUM; count++) {
                    sum = 0.0;
                    for (i = 0.0; i <= 3.0; i += 0.01) {
                                    x = sin(i); y = cos(i);
                                    sum += fabs(x*x + y*y - 1.0);
                    }
    }
    ticks = TickCount() - ticks;
    printf("Time      = %.2f seconds for %u iterations.\n",ticks/60.0,NUM);
    printf("Trig Err = %.18e\n",sum);
    printf("Loop Err = %.18e\n",i - 3.01);
    return(0);
}
```

End Listings

GENERAL PROGRAMMING RESOURCE GUIDE

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
AAIS Prolog	Advanced AI Systems	P.O. Box 39-0360	Mountain View, CA 94039	415-961-1121	$199.00
Adobe Illustrator	Adobe Systems	1585 Charleston Rd.	Mountain View, CA 94039	415-961-4400	$495.00
Afterthought	Jimmy Mac Software	P.O. Box 957	Murfreesboro, TN 37133	615-895-6427	$20.00
Allegro Common LISP	Coral Software	P.O. Box 307	Cambridge, MA 02142	617-547-2662	$600.00
Allegro add-ons	Coral Software	P.O. Box 307	Cambridge, MA 02142	617-547-2662	
ALS Prolog	Applied Logic Systems	P.O. Box 90, University Station	Syracuse, NY 13210-0090	315-471-3900	$349.00
Anatool	Advanced Logical Software	9903 Santa Monica Blvd., Ste. 108	Beverly Hills, CA 90212	714-390-0333	$925.00
APDA Online	CompuServe	5000 Arlington Ctr. Blvd., Box 20212	Columbus, OH 43220	800-848-8199	price varies
APL 68000, Version 6B	Spencer Organization	P.O. Box 248	Westwood, NJ 06765	201-666-6011	$99.00
APL*PLUS System	STSC Inc.	2115 E. Jefferson St.	Rockville, MD 20852	800-592-0500	$395.00
Apple 1MB Memory Expansion Kit	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	$249.00
Apple 2MB Memory Expansion Kit	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	$599.00
AutoDialog	Jam Technologies	685 Market St., Ste. 860	San Francisco, CA 94105	415-442-0795	$79.00
Aztec C Library Source	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$300.00
Aztec C Source Level Debugger	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$199.00
Aztec C68K/Mac c-Commercial	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$499.00
Aztec C68K/Mac d-Developer	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$299.00
Aztec C68K/Mac p-Professional	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$199.00
Aztec C68K-r c'Prime System	Manx Software Systems	One Industrial Wy.	Eatontown, NJ 07724	201-542-2121	$75.00
Basic Compiler	Microsoft	16011 NE 36th Way	Redmond, WA 98073-9717	206-828-8080	$195.00
Blue/20 System Flowcharts	Advanced Logical Software	9903 Santa Monica Blvd., Ste. 108	Beverly Hills, CA 90212	213-659-5157	$1,495.00
Blue/60	Advanced Logical Software	9903 Santa Monica Blvd., Ste. 108	Beverly Hills, CA 90212	213-659-5157	$1,495.00
BMUG (Macintosh User's Group)	BMUG Inc.	1442A Walnut St., Ste. 62	Berkeley, CA 94709		
B-Tree Helper 1.4	(M)agreeable Software	5925 Magnolia Ln.	Plymouth, MN 55442	612-559-1108	$75.00
BusTalk	Creative Solutions	4701 Randolph Rd., Ste. 12	Rockville, MD 20852	301-984-0262	$295.00
Canvas	Deneba Software	7855 NW St., Ste 202	Miami, FL 33126	305-594-6965	$195.00
CAPPS	Think Technologies	135 South Rd.	Bedford, MA 01730	617-275-4800	$75.00
CCSM (MacMUMPS)	MGlobal	1601 Westheimer, Ste. 201	Houston, TX 77006	713-529-4858	$149.95
CoCo Pro	Iconix Software Engineering	2800 28th St., Ste. 320	Santa Monica, CA 90405	213-458-0092	$495.00
Cognate	Peridom Inc.	P.O. Box 1812	Bowie, MD 20716	301-390-9570	$150.00
Cognate Developer's Kit	Peridom, Inc.	P.O. Box 1812	Bowie, MD 20716	301-390-9570	$250.00
ColorVueSE	Orchid Technology	45365 Northport Loop W	Fremont, CA 94538	415-683-0300	$695.00
Consulair 68K Development System	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$79.95
Consulair 68020/68881 C Compiler	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$600.00
C.P. Mac	Logique	30100 Town Center Dr., Ste. 198	Laguna Niguel, CA 92677	714-643-9147	$135.00
C Programmer's McTool Series (McCPrint)	MMC AD Systems	Box 360845	Milpitas, CA 95035	408-263-0781	$59.95
C Programmer's Toolbox, Volume I & II	MMC AD Systems	Box 360845	Milpitas, CA 95035	408-263-0781	$130.00
DBx	Desktop Ai	303 Linwood Ave.	Fairfield, CT 06430	203-255-3400	$550.00
Design	Meta Software	150 Cambridge Park Dr.	Cambridge, MA 02140	617-576-6920	$250.00
Design + DA Development System	Meta Software	150 Cambridge Park Dr.	Cambridge, MA 02140	617-576-6920	$2,000.00
Design Open Architecture Dev. Syst.	Meta Software	150 Cambridge Park Dr.	Cambridge, MA 02140	617-576-6920	$5,000.00
Desktop Help	Help Software	10659 A Maplewood Rd.	Cupertino, CA 95014	408-257-3815	$395.00
ED	Novasoft	2343 S. Ridgewood Ave.	Edgewater, FL 32032	904-423-7587	$1,500.00
ElectroFonts	Paragon Concepts	4954 Sun Valley Rd.	Del Mar, CA 92014	619-481-1477	$99.00
Eureka: The Solver	Borland International	4585 Scotts Valley Dr.	Scotts Valley, CA 95066	408-438-8400	$195.00
ExperCommon Lisp	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$995.00
ExperCommon Lisp II	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$1,195.00
ExperCommon Lisp Foreign Function Interface	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$295.00
ExperCommon OPS5	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$625.00
ExperFacts	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$495.00
ExperInterface Builder	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$395.00

GENERAL PROGRAMMING RESOURCE GUIDE

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
ExperLink	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$500.00
ExperLISP	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$495.00
ExperLisp File Compiler	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$395.00
ExperLogo	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$149.95
ExperLogo-Plus	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$95.00
ExperOPS+	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$495.00
ExperProlog II	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$495.00
Express Form	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$59.95
Extender Grafpak	Invention Software	6087 Jackson Rd.	Ann Arbor, MI 48106	313-996-8108	$69.95
Face It	Face Ware	1310 Broadway	Urbana, IL 61801	217-328-5842	$50.00
FreeForm for the Macintosh	Products Diversified	9720 Beechnut, #406	Houston, TX 77036	713-771-8357	$5,500.00
Guide	OWL International	14218 NE 21st St.	Bellevue, WA 98007	206-747-3203	$134.95
HeapShow	B/T Computing	P.O. Box 1465	Euless, TX 76039	817-267-1415	$79.00
Humble Expert System Shell	Xerox Special Information Systems	250 N. Halstead St.	Pasadena, CA 91107	818-351-2351	$400.00
Iconix PowerTools Version 2.0	Iconix Software Engineering	2800 28th St., Ste. 320	Santa Monica, CA 90405	213-458-0092	$995-$4,495
IFAM	Tech-Pro Utilities/APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	
Inside Macintosh X-Ref on Disk	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$10.00
Instant Expert	Human Intellect Systems	1670 S. Amphlett Blvd., Ste. 326	San Mateo, CA 94402	415-571-5939	$49.95
Instant Expert Plus	Human Intellect Systems	1670 S. Amphlett Blvd., Ste. 326	San Mateo, CA 94402	415-571-5939	$498.00
ISAM4	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for Lightspeed C	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for Lightspeed Pascal	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for Microsoft BASIC	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for MPW C	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for MPW Pascal	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
ISAM/SORT for Turbo Pascal	Rocinate	P.O. Box 55427	Seattle, WA 98155	206-365-0897	$89.95
Knowledge Index	Dialog Information Services	3460 Hillview Ave.	Palo Alto, CA 94304	800-334-3564	
LabVIEW	National Instruments	12109 Technology Blvd.	Austin, TX 78727-6204	512-250-9119	$1,995.00
Language Systems FORTRAN	Language Systems Corp.	463 Carlisle Dr.	Herndon, VA 22070	703-478-0181	$295.00
LANSTAR AppleTalk	Northern Telecom	2305 Mission College	Santa Clara, CA 95054	408-988-5550	
Laser FX	PostCraft	27811 Ave. Hopkins, Ste. 6	Valencia, CA 91355	805-257-1797	$195.00
Lasertalk	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$210.50
LeLisp	ExperTelligence	559 San Ysidro Rd.	Santa Barbara, CA 93108	805-969-7874	$295.00
Lightspeed C	Think Technologies	135 South Rd.	Bedford, MA 01730	617-275-4800	$175.00
Lightspeed Pascal	Think Technologies	135 South Rd.	Bedford, MA 01730	617-275-4800	$125.00
Linear and Non-Linear Programming	Lionheart Press	P.O. Box 379	Alburg, VT 05440	514-933-4918	$95.00
LogarithMac	Heizer Software	5120 Coral Ct.	Concord, CA 94521	415-827-9013	$39.95
Logic Compiler	Capilano Computing Systems	501-1168 Hamilton St.	Vancouver, BC Canada V6B 2S2	604-669-6343	$299.95
LogiFonts	Paragon Concepts	4954 Sun Valley Rd.	Del Mar, CA 92014	619-481-1477	$40.00
LPA MacProlog (Wizard Edition)	Logic Programming Associates	Studio 4, RVPB, Trinity Rd.	London, England SW18 3SX	1-871-2016	$495.00
LPA MacProlog (Student Edition)	Logic Programming Associates	Studio 4, RVPB, Trinity Rd.	London, England SW18 3SX	1-871-2016	$275.00
LPLC	Capilano Computing Ltd.	501-1168 Hamilton St.	Vancouver, BC Canada V6B 2S2	604-669-6343	$299.95
Mac II Tools	Creative Solutions	4701 Randolph Rd., Ste. 12	Rockville, MD 20852	301-984-0262	$79.00
Mac286	AST Research	2332 McGaw Ave.	Irvine, CA 92714	714-553-0340	$1,599.00
Mac+PC Kit, The Hardware	PerfecTEK Corp.	726 S. Hillview Dr.	Milpitas, CA 95035	408-263-7757	$995.00
MacAdvantage 68000 Assembler	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$99.95
MacAdvantage UCSD Pascal, Version 2.3	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$99.95
MacAPL, Verison 2.21	Leptonic Systems	405 Tarrytown Rd., #145	White Plains, NY 10607	914-682-0377	$125.00
MacApp, Version 1.1.1	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$100.00
Mac Apple User's Group (MAUG)	CompuServe	5000 Arlington Ctr. Blvd., Box 20212	Columbus, OH 43220	800-848-8199	
MacApp Object Library #1	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$15.00
MacApp Object Library #2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$25.00
MacApp Object Library #3	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$25.00
MacApp Object Library #4	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$45.00
MacApp Object Library #5	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$40.00
MacApp Source Listings	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$30.00
MacAsm	Mainstay	5311-B Derry Rd.	Agoura Hills, CA 91301	818-991-6540	$125.00
MacBrain, Version 1.10a	Neuronics Inc.	One Kendall Sq., Ste. 2200	Cambridge, MA 02139	617-577-1202	
Mac C and Mac C Toolkit	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$425.00
MacC Jr.	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$79.95
MacDesigner	Excel Software	P.O. Box 1414	Marshalltown, IA 50158	515-752-5359	$498.00
MacEmacs	UniPress Software	2025 Lincoln Hwy.	Edison, NJ 08817	201-985-8000	$249.00
Mac Example Applications/ Sources 1.0	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$15.00
MacExpress	ALSoft	P.O. Box 927	Spring, TX 77383	713-353-4090	$195.00
MacFlow 2.0	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$195.00
MacForth Plus, Version 3.52	Creative Solutions	4701 Randolph Rd.	Rockville, MD 20852	301-984-0262	$199.00
MacForth Plus Tools Disk #1	Creative Solutions	4701 Randolph Rd.	Rockville, MD 20852	301-984-0262	$30.00
MacForth 3-D Library	Creative Solutions	4701 Randolph Rd.	Rockville, MD 20852	301-984-0262	$25.00
MacFORTRAN	Absoft	2781 Bond St.	Auburn Hills, MI 48057	313-853-0050	$295,$495
Mac to GS Assembler/Linker	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$195.00
Mach II	Palo Alto Shipping	P.O. Box 7430	Menlo Park, CA 94026	415-363-1399	$99.95
MacinTalk Development Pkg.	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$10.00
Macintosh 68000 Development System	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	$195.00
Macintosh Ada Compiler	Meridian Software	23141 Verdugo Dr., Ste. 105	Laguna Hills, CA 92653	714-380-9800	$1,195.00
Macintosh Consultants Network	Match Data Systems			206-453-2951	
Macintosh Development Utilities	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$25.00
Macintosh Introductory Programming	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$49.95
Macintosh Pascal	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	$125.00
Macintosh Power System	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$99.95
Macintosh Programmer's Workshop	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$200.00
The Macintosh Power System	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$79.95
Macintosh Reference System	TOM Programs	1500 Massachusetts Ave. NW, Ste. 34-MG	Washington, DC 20005	202-223-6813	$125.00

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
Macintosh System Software 1.0-5.0	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$15-$17.50
MacLink Plus	Dataviz Inc.	16 Winfield St.	Norwalk, CT 06855	203-866-4944	$195.00
MacMETH 2.0	Modula Corp.	950 N. University Ave.	Provo, UT 84604	801-224-8999	$100.00
MacModula-2 4.1	Modula Corp.	950 N. University Ave.	Provo, UT 84604	801-224-8999	$150.00
MacMUMPS	MGlobal	1601 Westheimer, Ste. 201	Houston, TX 77006	713-529-4858	$149.95
MacNET	MacNET	10101 Bubb Rd.	Cupertino, CA 95014	408-973-0110	$49.95
MacNosy and The Debugger 2	Jasik Designs	343 Trenton Wy.	Menlo Park, CA 94025	415-322-1386	price varies
Macomo	OITC Inc.	P.O. Box 73	Melbourne Beach, FL 32951	305-984-3714	$185.00
Mac-PROLOG	Programming Logic Systems	31 Crescent Dr.	Milford, CT 06460	203-877-7988	$295.00
Mac Reference System 128	TOM Programs	1500 Massachusetts Ave. NW, Ste. 34-R	Washington, DC 20005	202-223-6813	$125.00
MacScheme	Semantic Microsystems	4470 S.W. Hall St., #340	Beaverton, OR 97005	503-643-4539	$125.00
MacScheme + Toolsmith	Semantic Microsystems	4470 S.W. Hall St., #340	Beaverton, OR 97005	503-643-4539	$395.00
Mac Skills	ComputerWare Learning Center	490 California Ave.	Palo Alto, CA 94306	415-323-7951	
Mac Skills: Beyond the Basics	ComputerWare Learning Center	490 California Ave.	Palo Alto, CA 94306	415-323-7951	
MacSMARTS 3	Cognition Technology Corp.	55 Wheeler St.	Cambridge, MA 02138	617-492-0246	$149.95
MacSnap 1024	Dove Computer Corp.	1200 N. 23rd St.	Wilmington, NC 28405	919-763-7918	$299.00
MacSnap 4S	Dove Computer Corp.	1200 N. 23rd St.	Wilmington, NC 28405	919-763-7918	$1,495.00
MACtoGS Assembler/Linker	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$195.00
Mactran Plus 3.0	DCM Data Products	1710 Two Tandy Center	Ft. Worth, TX 76102	817-870-2202	$399.00
MacTRAN 77 Fortran 2.0	DCM Data Products	1710 Two Tandy Center	Ft. Worth, TX 76102	817-870-2202	$199.00
MacΣqn	Software for Recognition Technology	55 Academy Dr.	Rochester, NY 14623	716-359-3024	$44.95
Mail Order Products	Central Point Software	9700 SW Capitol Hwy., #100	Portland, OR 97219	503-244-5782	
Mansion Bulletin Board	Michael Pester	667 49th St.	Des Moines, IA 50312	515-279-9650	$49.95
Mass Pak	Mass Micro Systems	550 Del Rey	Sunnyvale, CA 94086	408-988-1200	$549.00
MasterFORTH	MicroMotion	8726 S. Sepulveda Blvd., No. A171	Los Angeles, CA 90045	213-821-4348	$299.00
MathLab	E & M Software Co.	95 Richardson Rd.	North Chelmsford, MA 01863	617-251-7451	$49.95
MathView Professional	BrainPower Inc.	24009 Ventura Blvd.	Calabasas, CA 91302	818-884-6911	$249.95
Matrix Operations	Lionheart Press Inc.	P.O. Box 379	Alburg, VT 05440	514-933-4918	$95.00
McFace Sub	Tensor Laboratories	P.O. Box 9723	Stanford, CA 94305	707-763-7873	$40.00
McFace Tools	Tensor Laboratories	P.O. Box 9723	Stanford, CA 94305	707-763-7873	$119.00
Megamax C	Megamax	P.O. Box 851521	Richardson, TX 75085		$299.95
Memory Map	SoftPlus	14500 Chrisman Hill Dr.	Boyds, MD 20841	301-540-6552	$59.95
MicroExplorer	Texas Instruments	2501 S. Hwy. 121	Dallas, TX 75067	214-462-4111	
MicroSlate	Preceptor Systems	P.O. Box 2803	Columbia, MD 21045	301-730-3401	$150.00
Microsoft BASIC Compiler	Microsoft	16011 NE 36th Wy.	Redmond, WA 98073-9717	206-882-8080	$195.00
Microsoft BASIC Interpreter	Microsoft	16011 NE 36th Wy.	Redmond, WA 98073-9717	206-882-8080	$99.00
Microsoft Fortran 2.2	Microsoft	16011 NE 36th Wy.	Redmond, WA 98073-9717	206-882-8080	$295.00
MIDIBasic	Altech Systems	831 Kings Hwy.	Shreveport, LA 71119	318-226-1702	$49.95
MiniLogo	Viking Technologies	174 Bellevue Ave.	Newport, RI 02840	800-437-0033	$15.95
MPW 2.0.2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$200.00
MPW Assembler 2.0.2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$100.00
MPW C 2.0.2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$150.00
MPW Enhancer 1.0	Sand Hill Engineering c/o APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$35.95;$43.95
MPW Pascal 2.0.2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$150.00
MS-DOS Board	PerfecTEK Corp.	1455 McCarthy Blvd.	Milpitas, CA 95035	408-263-7757	$995.00
MultiSIMMS	MacMemory	2480 N. First St.	San Jose, CA 95131	408-922-0140	$1,799.00
Neon	Kriya Systems	Six Export Dr.	Sterling, VA 22170	800-345-7492	$50.00
Nexpert Object	Neuron Data	444 High St.	Palo Alto, CA 94301	415-321-4488	$800-$5000
Nexus	Human Intellect Systems	1670 S. Amphlett Blvd., Ste. 326	San Mateo, CA 94402	415-571-5939	$4,800.00
Nibble Mac Utility Pack	MicroSparc/Nibble Publications	52 Domino Dr.	Concord, MA 01742	617-371-1660	$29.95
NS8/16 Memory Expansion Board for the Mac	National Semiconductor	P.O. Box 58090	Santa Clara, CA 95052-8090	408-721-5000	
Numerical Recipes	Numerical Recipes Software	P.O. Box 243	Cambridge, MA 02238		$39.00
Object LOGO	Coral Software	P.O. Box 307	Cambridge, MA 02142	617-547-2662	$79.95
OPS5-+	Computer*Thought Corp.	840 Ave. F., Ste. 104	Plano, TX 75074	214-424-3511	$1,800.00
PCMacBASIC	Pterodactyl Software	905 W. California St.	Mill Valley, CA 94941	415-388-4827	$39.95
Pearl Lisp	Coral Software	P.O. Box 307	Cambridge, MA 02142	617-547-2662	$99.00
Personal Prolog	Optimized Systems Software	221-B Kentwood Ave.	San Jose, CA 95129	408-446-3099	$74.95
Phoenix Macintosh SIG	Portal Communications Co.	19720 Auburn Dr.	Cupertino, CA 95014	408-973-9111	
PICT Detective	Palomar Software	P.O. Box 2635	Vista, CA 92083	619-727-3922	$69.95
Polytron Version Control System (PCVS)	Polytron Corp.	1815 NW 169th Pl., Ste. 2110	Beaverton, OR 97006	503-645-1150	$149,$395
PostHaste	Micro Dynamics Ltd.	8555 16th St., Ste. 802	Silver Spring, MD 20910	301-589-6300	$59.95
Postscript	Adobe Systems	1585 Charleston Rd.	Mountain View, CA 94039	415-961-4400	varies
Professional Pack	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$249.95
Professional Programmer's Extender	*Invention Software*	P.O. Box 3168	Ann Arbor, MI 48106	313-996-8108	$395.00
Program Development Tools and Utilities	Pecan Software Systems	1410 39th St.	Brooklyn, NY 11218	718-851-3100	$39.95 - $99.95
Programming the Macintosh II: Volumn I	Personal Concepts	635 Wellsbury Wy.	Palo Alto, CA 94036		
Programmer's Extender vol. 1 & 2	Invention Software	P.O. Box 3168	Ann Arbor, MI 48106	313-996-8108	$89.95
Programmer's On-Line Companion 2.0	Addison-Wesley	Rte. 128	Reading, MA 01867	617-944-3700	$49.95
Prograph	TGS Systems	1127 Barrington St., Ste. 19	Halifax, NS Canada B3H 2P8	902-429-5642	$99.00
Prolog II	Avenue Software	1173 West Charest Blvd., Ste. #250	Quebec City, PQ Canada G1N 2C9	418-682-3088	$395.00
Prolog/m	Chalcedony Software	5580 La Jolla Blvd.	La Jolla, CA 92037	619-483-8513	$99.95
Prototyper	SmethersBarnes	P.O. Box 639	Portland, OR 97207	800-237-3611	$125.00
P-tral	Woodchuck Industries	340 W 17th St., #2B	New York, NY 10011	212-629-5617	$99.95
QUED	Paragon Concepts	4954 Sun Valley Rd.	Del Mar, CA 92014	619-481-1477	$65.00
QUED/m	Paragon Concepts	4954 Sun Valley Rd.	Del Mar, CA 92014	619-481-1477	$119.00
Radius Accelator 15	Radius Inc.	404 E. Plumeria Dr.	San Jose, CA 95134	408-434-1010	
Radius Accelator 25	Radius Inc.	404 E. Plumeria Dr.	San Jose, CA 95134	408-434-1010	

GENERAL PROGRAMMING RESOURCE GUIDE

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
Radius Full Page Display	Radius Inc.	404 E. Plumeria Dr.	San Jose, CA 95134	408-434-1010	$1,995.00
Ram I	Open Mac Enterprises	2280 Bates Ave., Ste. J	Concord, CA 94520	415-682-0440	$199.00
Ram II+II	Open Mac Enterprises	2280 Bates Ave., Ste. J	Concord, CA 94520	415-682-0440	$999.00
Rascal II	Metaresearch	1211 SW Fifth, #2860	Portland, OR 97204	503-228-5806	$149.00
RasterOps 24-bit ColorBoard	RasterOps	10161 Bubb Rd.	Cupertino, CA 95014	408-446-4090	$2,795.00
RBasic	Indexed Software	40960 E. Florida Ave.	Hemet, CA 92344	714-929-5707	$149.00
Reactions	E & M Software Co.	95 Richardson Rd.	North Chelmsford, MA 01863	617-251-7451	$28.00
Red Ryder Host	The FreeSoft Company	10828 Lacklink	St. Louis, MO 63114	314-423-2190	$60.00
Scientific Symbols Fonts	Probability Distribution	P.O. Box 27276	Austin, TX 78755-2276	512-338-1250	$40.00
SciFonts	Paragon Concepts	4954 Sun Valley Rd.	Del Mar, CA 92014	619-481-1477	$50.00
SCSI Development Package 1.0	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$10.00
SCSI Tool	Arborworks Inc.	1810 Sunrise St.	Ann Arbor, MI 48103	313-747-7087	
SemperSoft Modula-2	Semper Software	P.O. Box 225	Glen Ellyn, IL 60138	312-790-1253	$125.00
Simply-Done Charts and Graphs	SouthWest Software Design and Consulting	P.O. Box 23795	Tempe, AZ 85282		$49.00
Simply-Done I/O and Simply-Done Numerics	SouthWest Software Design and Consulting	P.O. Box 23795	Tempe, AZ 85282		$49.00
Simply-Done Measurement Displays and Controls	SouthWest Software Design and Consulting	P.O. Box 23795	Tempe, AZ 85282		$39.00
Smalltalk-80 0.4	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$75.00
Smalltalk-80 CL	ParcPlace Systems	2400 Geng Rd.	Palo Alto, CA 94303	415-859-1000	$695.00
Smalltalk-80 DE	ParcPlace Systems	2400 Geng Rd.	Palo Alto, CA 94303	415-859-1000	$995.00
SuperExpert	Softsync Inc.	162 Madison Ave.	New York, NY 10016	212-685-2080	$199.95
Switcher Developer's Kit1.0	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$13.50
Tags	MicroMotion	8726 S. Sepulveda Blvd., #A171	Los Angeles, CA 90045	213-821-4348	$495.00
Tape/Disk Conversions	Pivar Computing Services	165 Arlington Heights Rd.	Buffalo Grove, IL 60089	312-459-6010	varies
TechnoFile	Jam Technologies	685 Market St., Ste. 860	San Francisco, CA 94105	415-442-0795	$49.00
Teleport 3.33	Stan Krute's Hacker and Nerd	18617 Camp Creek Rd.	Hornbrook, CA 96044	916-475-3428	$30.00
Terrapin Logo	Terrapin	376 Washington St.	Malden, MA 02148	617-492-8819	$99.95
TexSys	ToolMasters, Ltd.	600 Herndon Pkwy.	Herndon, VA 22070	703-478-9808	$69.00
TML Data Base Toolkit	TML Systems	4241 Bay Meadows Rd., Ste. 23	Jacksonville, FL 32217	904-636-8592	$89.95
TML Modula-2	TML Systems	4241 Bay Meadows Rd., Ste. 23	Jacksonville, FL 32217	904-636-8592	$99.95
TML MacLanguage Series-Pascal	TML Systems	4241 Bay Meadows Rd., Ste. 23	Jacksonville, FL 32217	904-636-8592	$99.95
TML Source Code Library	TML Systems	4241 Bay Meadows Rd., Ste. 23	Jacksonville, FL 32217	904-636-8592	$79.95
TMON Debugger	Icom Simulations	626 S. Wheeling Rd., Ste. 10	Wheeling, IL 60090	312-520-4440	$149.95
Toolbox Interface Reference Pkg. 2.0.2	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$12.00
TrueBASIC1.2	True BASIC Inc.	39 S. Main St.	Hanover, NH 03755	603-643-3882	$99.95
TrueBASIC Advanced String Library	True BASIC Inc.	39 S. Main St.	Hanover, NH 03755	603-643-3882	$49.95
TrueBASIC Communications Support Library	True BASIC Inc.	39 S. Main St.	Hanover, NH 03755	603-643-3882	$49.95
TrueBASIC Macintosh Runtime Package	True BASIC Inc.	39 S. Main St.	Hanover, NH 03755	603-643-3882	$99.95
TrueBASIC Sorting & Searching Library	True BASIC Inc.	39 S. Main St.	Hanover, NH 03755	603-643-3882	$49.95
True Color & RasterOps 24-bit ColorBoards	RasterOps	10161 Bubb Rd.	Cupertino, CA 95014	408-446-4090	$2,795.00
Turbo Pascal for the Mac	Borland International	4585 Scotts Valley Dr.	Scotts Valley, CA 95066	408-438-8400	$99.95
Turbo Pascal Database Toolbox	Borland International	4585 Scotts Valley Dr.	Scotts Valley, CA 95066	408-438-8400	$99.95
Turbo Pascal Numerical Methods Toolbox	Borland International	4585 Scotts Valley Dr.	Scotts Valley, CA 95066	408-438-8400	$99.95
Turbo Pascal Tutor	Borland International	4585 Scotts Valley Dr.	Scotts Valley, CA 95066	408-438-8400	$69.95
Universal Cross Assembler	Memocom Development Tools	Arbor Creek Dr.	Carrollton, TX 75010	214-446-9906	$299.00
Universal Interface Converter	Jonathan Freeman Design	P.O. Box 880114	San Francisco, CA 94188	415-822-8451	$119.00
Utilities (4 dev. system)	Consulair Corp.	140 Campo Dr.	Portola Valley, CA 94025	415-851-3272	$50.00
Visual Interactive Programming (VIP)	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$124.95
VIP Advanced Manager	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$95.00
VIP to Lightspeed C Translator	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$124.95
VIP to Lightspeed Pascal Translator	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$124.95
VIP to MPW Pascal Translator	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$89.95

(continued from page 35)
generates the first 1,899 prime numbers by a method attributed to Eratosthenes. Table 4, below, gives the results.

Version 1.1 of the Dhrystone benchmark represents the mix of instructions that the average C program contains. It is therefore 53 percent assignments, 32 percent control statements, and 15 percent procedure calls. This Usenet favorite (see Listing Two, page 27) has been run on almost every machine made. All the results are displayed in the traditional Dhrystones-per-second notation, but the tests were done with 50,000 iterations of the code. Table 5, below, gives the results.

The Sqrt benchmark (see Listing Three, page 32) is similar to the Sieve for floating point: It is one loop that adds the square roots of the first 100,000 integers. Table 6, below, gives the results.

The Hilbert benchmark (see Listing Four, page 33) tests basic floating-point arithmetic, both in speed and accuracy. This benchmark, from Dr. Paul Finlayson of the Apple Numerics Group, generates a 10 × 10 Hilbert matrix whose determinant is on the order of 10–21. A related matrix equation is then solved by Gaussian elimination. The resulting solution vector has a result of all 1s if the arithmetic is exact. The total error is a measure of the vector's deviation from the exact value. Table 7, below, gives the results.

The Trig benchmark (see Listing Five, page 34) is another useful benchmark by Dr. Finlayson that tests the speed and accuracy of an implementation's trigonometric functions by calculating the Pythagorean identity every 0.01 radians from zero to three. The inner loop should therefore be executed 300 times. Note that in this case the trig error for the direct calls is an order of magnitude worse than for the software calls.[15] Table 8, below, gives the results.

Documentation

Several levels of help are provided with MPW. An on-line Help command gives concise summaries of commands and their options, expression syntax and precedence, selections, and keyboard shortcuts. Any user of MPW can modify and extend the help system to include additional information.

Parameter	MPW C	MPW Pascal
Build time (sec)	00,009	00,010
Time for 100 iter. (sec)	00,006.37	00,011.97
Object code size (bytes)	00,244	00,406
Exec. code size (bytes)	14,343	13,191

Table 4: *Sieve benchmark results*

Software SANE	MPW C	MPW Pascal
Build time (sec)	00,015	00,011
Dhrystones/sec	02,173	02,777
Object code size (bytes)	01,440	01,832
Exec. code size (bytes)	16,385	15,541
Direct to 68881	**MPW C**	**MPW Pascal**
Build time (sec)	00,016	00,011
Dhrystones/sec	02,500	02,941
Object code size (bytes)	01,420	01,832
Exec. code size (bytes)	16,579	15,405

Table 5: *Dhrystone benchmark results*

Software SANE	MPW C	MPW Pascal
Build time (sec)	00,010	00,009
Execution time (sec)	00,043.65	00,040.18
Object code size (bytes)	00,312	00,314
Exec. code size (bytes)	14,461	2,815
Direct to 68881	**MPW C**	**MPW Pascal**
Build time (sec)	00,009	00,009
Execution time (sec)	00,001.48	00,002.17
Object code size (bytes)	00,164	00,290
Exec. code size (bytes)	04,465	12,793

Table 6: *Sqrt benchmark results*

Software SANE	MPW C	MPW Pascal
Build time (sec)	00,014	00,011
Time for 100 iter. (sec)	00,016.72	00,015.00
Object code size (bytes)	01,760	01,670
Exec. code size (bytes)	16,171	14,337
Total error	4.435 E-7	4.435 E-7
Direct to 68881	**MPW C**	**MPW Pascal**
Build time (sec)	00,013	00,010
Time for 100 iter. (sec)	00,001.68	00,001.75
Object code size (bytes)	01,192	01,392
Exec. code size (bytes)	15,643	14,061
Total error	4.435 E-7	4.435 E-7

Table 7: *Hilbert benchmark results*

Software SANE	MPW C	MPW Pascal
Build time (sec)	00,010	00,010
Time for 100 iter. (sec)	00,180.28	00,173.57
Object code size (bytes)	00,512	00,636
Exec. code size (bytes)	14,821	13,179
Trig error	4.662 E18	4.662 E-18
Direct to 68881	**MPW C**	**MPW Pascal**
Build time (sec)	00,009	00,008
Time for 100 iter. (sec)	00,003.27	00,003.70
Object code size (bytes)	00,266	00,516
Exec. code size (bytes)	14,683	13,019
Trig error	4.597 E17	4.597 E17

Table 8: *Trig benchmark results*

(continued from page 39)

An MPW tool called Commando is a second level of help, oriented toward building command lines. Commando (written by Tom Taylor in his spare time!) provides a Mac-like dialog interface to tools and was new with MPW 2.0. Any MPW tool can be made Commando compatible by adding a single *cmdo* resource to its resource fork. Typing a command name followed by the ellipsis character invokes a dialog that offers radio buttons, pop-up menus, check boxes, text fields, and on-line explanations of each option for that tool. The resulting command line is displayed dynamically in the dialog box and can then be copied or immediately executed. Commando is a great way to explore new options and to learn about tools. Every command in the MPW system has been Commando-ized for the 2.0 release, even the built-ins. Commando saves frequent forays into the manual pages.

A third level of help comes through MPW shell scripts that automate and guide you through building an application, desk accessory, or tool. These powerful scripts are collectively called build scripts. The build scripts install two additional menus in the menu bar, which allow you to change directories quickly and to generate and run make files automatically. These scripts were written by Rick Meyers to help beginning users with the otherwise potentially overwhelming system, but they are also useful to anyone who needs to crank out a tool quickly. Each language (Asm, C, or Pascal) includes three programs to build and play with using the build scripts: a sample Mac application, a sample desk accessory, and a sample integrated tool.

The main MPW reference manual fully describes the shell and its commands. It has two parts: the shell tutorial and the manual pages for the commands and tools.

Notes

1. Note that the C and Pascal products are sold separately; Pascal includes the Pascal-specific tools. The items listed in Table 3 are MPW tools except for ResEdit, which is a stand-alone Mac application, and MacsBug, which is a debugger. This list is for MPW 2.0. In addition to this list are a few conversion tools useful to those bringing forward sources from the Lisa Workshop and MDS.
2. *Macintosh Programmer's Workshop Assembler Reference* (Renton, Wash.: APDA, 1987).
3. This benchmark is derived from assembling the 68xxx disassembler in MacsBug, in which 8,033 lines of as-

MPW Background, History, and Credits

Development on MPW began late in 1984, when Apple engineer Rick Meyers was assigned to bring about a development environment to suit Apple's internal requirements.

The original team consisted of three people: Rick Meyers, Jeff Parrish, and Dan Smith. The project was originally known as MPS, for Macintosh Programming System (not Meyers, Parrish, and Smith!). All three had worked on major Lisa software projects: Rick worked on C and Smalltalk, Jeff worked on MacWorks, and Dan wrote the Lisa Finder. Together these three engineers began work on the core application of MPW, the MPW shell. They began their effort by porting the MDS Edit program, which had been written for Apple in C by Bill Duvall of Consulair Corp. Much of the early work on the MPW shell was done in C on three Apollo workstations. As the Lisa Workshop's Green Hills C compiler was ported to run under MPW, so the development of the MPW shell moved to Macintosh.

As the design of the MPW shell progressed, a need for two different applications became apparent: a Unix-like command shell and a Mac-like mouse-based editor. Further, it became clear that these two applications needed to be tightly coupled. The solution was a combination shell/editor. Dan Smith wrote the shell and Jeff Parrish wrote the editor. Project leader Rick Meyers worked on the command interpreter.

By early 1985, others had joined the effort, most of them also key contributors to the previous Lisa Workshop. Ira Ruben wrote a completely new 68xxx assembler from scratch as well as many of the other tools for MPW. Fred Forsman wrote two major utilities: Make and Print. Ken Friedenbach brought the Lisa Linker forward, with major enhancements. Jim Thomas (head of the Development Systems group) and Clayton Lewis (of the Numerics group) made sure that the Standard Apple Numerics Environment (SANE) was implemented properly across all the languages. (For more information on SANE, see the *Apple Numerics Manual*, Addison-Wesley, 1986. The second edition, due this year, covers the 68881 details of SANE.)

Johan Strandberg designed the *rez* line of resource tools, and Tom Taylor finished them. Gene Pope worked on *resedit*. Neal Johnson supervised the equates files and Steve Hartwell (formerly of Bell Labs) ported the standard C libraries. Mike Shannon took over dealing with the Green Hills C compiler from Rick Meyers, and Al Hoffman, Key Doyle, and Roger Lawrence brought Lisa Pascal forward. Russ Daniels began work on a symbolic debugger. When that became too ambitious for the schedule, Dan Allen joined the team and rewrote MacsBug, the Macintosh assembly level debugger. Russ contributed greatly as Chief Heap Dump Analyzer for the group.

Chris Brown and at least 13 other people worked on testing the system, as did many beta testers. Paul Zemlin was the product manager, Harry Yee built the system for Apple's Software Configuration Management (SCM) group, and Lita Parr kept the whole group coordinated and running smoothly.

It took a year and a half to create MPW 1.0, which began distribution through the Apple Programmer's and Developer's Association (APDA) in September 1986. The team then grew substantially and released MPW 2.0 through APDA in July 1987. Work on MPW 3.0 is currently underway with an even bigger team. —D.A.

sembly-language code are assembled on a Mac II in 11.4 seconds, or at a rate of 42,279 lines per minute. A Mac Plus would be about one quarter as fast.

4. Samuel P. Harbison and Guy L. Steele, Jr., *C: A Reference Manual* (Englewood Cliffs, N.J.: Prentice-Hall, 1987).

5. AT&T Bell Labs and M. I. Bolsky, *The C Programmer's Handbook* (Englewood Cliffs, N.J.: Prentice-Hall, 1985).

6. *Macintosh Programmer's Workshop C Reference* (Renton, Wash.: APDA, 1987). 1987).

7. *Units* originated with UCSD Pascal. Other versions of Pascal for the Macintosh have since been enhanced with the addition of *Units*, including Turbo Pascal from Borland and LightSpeed Pascal from Think Technologies. Turbo Pascal 5.0 for the IBM PC now also has *Units*.

8. Brian W. Kernighan, "Why Pascal Is Not My Favorite Programming Language," Bell Laboratories internal memo #81-11272-12 (April 2, 1981).

9. *Macintosh Programmer's Workshop Pascal Reference* (Renton, Wash.: APDA, 1987).

10. At least two versions of Modula-2 are currently available and at least two versions of FORTRAN are under development. There is a growing market for further languages and tools to be developed and marketed for MPW.

11. Benchmarks performed on a Mac II with an Apple HD-80SC drive and MPW running in 1 Mbyte under MultiFinder.

12. Bill Atkinson's HyperCard and MacPaint programs are about 95 percent Pascal object code and 5 percent assembly-language object code.

13. This equation is from Peter Wegner's paper entitled "Dimensions of Object-Based Language Design," given at OOPSLA '87.

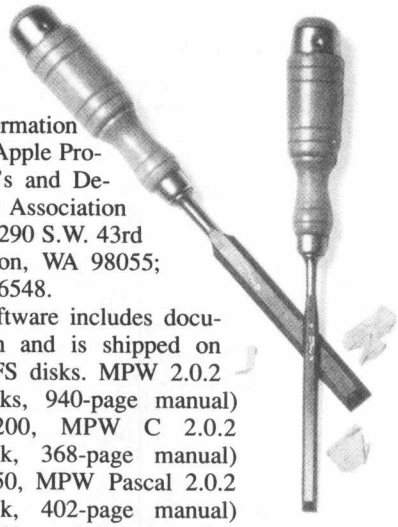
14. For more information about data abstraction and object-oriented programming from a language-design point of view, see *Programming Language Concepts*, by Carlo Ghezzi and Mehdi Jazayeri (New York: Wiley, 1987).

15. Most of the Motorola 68881 functions return identical results as the software versions, but the elementary functions are an exception. Motorola traded accuracy for speed. The *–elems881* compiler switch allows users to decide which set of routines are called: the hardware or the software routines.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

MPW and MacApp are available through the Apple Programmer's and Developer's Association (APDA). For more information contact: Apple Programmer's and Developer's Association (APDA) 290 S.W. 43rd St., Renton, WA 98055; 206-251-6548.



All software includes documentation and is shipped on 800K HFS disks. MPW 2.0.2 (five disks, 940-page manual) costs $200, MPW C 2.0.2 (one disk, 368-page manual) costs $150, MPW Pascal 2.0.2 (one disk, 402-page manual) costs $150, and MacApp 1.1.1 (two disks, 470-page manual) costs $100.

MPW 1.0 supported development on any Macintosh with 1-Mbyte RAM and at least 1.6 Mbytes of disk space and was shipped on 400K floppies. MPW 2.0 ships on 800K floppies and requires 128K ROM (or more) and a hard disk. MPW 2.0 does not support the Mac XL, but MPW 1.0 is still available through APDA for those with Mac XLs.

DDJ

Dan Allen is a software explorer for Apple Computer where he has worked on several projects, including MacApp, HFS, MacPlus, MacsBug, and MPW. He is currently working with Bill Atkinson on HyperCard. Dan can be reached at Apple Computer Inc., 20525 Mariani Ave., MS: 27E, Cupertino, CA 95014.

BIBLIOGRAPHY

Technical Documents

1985 Macintosh Technical Notes APDA: Renton, Wash.

1986 Macintosh Technical Notes. APDA: Renton, Wash.

1987 Macintosh Technical Notes. APDA: Renton, Wash.

Apple II & Macintosh Development Software: A Guide to Languages & Tools—Summer 1987. APDA: Renton, Wash.

Apple File Exchange Technical Reference, Version 1.1. APDA: Renton, Wash.

Apple Publications Style Guide. APDA: Renton, Wash.

MacApp 1.1 Source Listings. APDA: Renton, Wash.

Macintosh Family Hardware Reference, Draft. APDA: Renton, Wash.

Macintosh SE Technical Tour. APDA: Renton, Wash.

MultiFinder Development Package, Version 1.0. APDA: Renton, Wash.

PICT File Format Notes. APDA: Renton, Wash.

PICT File Format Notes Disk Set. APDA: Renton, Wash.

SCSI Development Package, Version 1.0. APDA: Renton, Wash.

Software Applications in a Shared Environment. APDA: Renton, Wash.

Books and Periodicals

Adobe Systems Inc. *PostScript Language Program Design*. Addison-Wesley: Reading, Mass. 1988.

Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley: Reading, Mass. 1985.

Adobe Systems Inc. *PostScript Tutorial and Cookbook*. Addison-Wesley: Reading, Mass. 1985.

Apple Computer Inc. *Apple Numerics Manual*. Addison-Wesley: Reading, Mass. 1987.

Apple Computer Inc. *Apple Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley: Reading, Mass. 1988.

Apple Computer. *Designing Cards and Drivers for Macintosh*

Apple Computer Inc. *Inside Macintosh, Vols. 1–3*. Addison-Wesley: Reading, Mass. 1985.

Apple Computer Inc. *Inside Macintosh, Vol. 1*. Addison-Wesley: Reading, Mass. 1985.

Apple Computer Inc. *Inside Macintosh, Vol. 2*. Addison-Wesley: Reading, Mass. 1985.

Apple Computer Inc. *Inside Macintosh, Vol. 3*. Addison-

(continued from page 41)

Wesley: Reading, Mass. 1985.

Apple Computer Inc. *Inside Macintosh, Vol. 4*. Addison-Wesley: Reading, Mass. 1986.

Apple Computer Inc. *Inside Macintosh, Vol. 5*. Addison-Wesley: Reading, Mass. 1988.

Apple Computer Inc. *Inside Macintosh X-REF*. Addison-Wesley: Reading, Mass. 1988.

Apple Computer Inc. *Macintosh Family Hardware Reference*. Addison-Wesley: Reading, Mass. 1988.

Apple Computer Inc. *Programmer's Introduction to the Macintosh Family*. Addison-Wesley: Reading, Mass. 1986.

Apple Computer Inc. *Technical Introduction to the Macintosh Family*. Addison-Wesley: Reading, Mass. 1987.

Chernicoff, Stephen. *The Macintosh Revealed, Volume One: Unlocking the Toolbox, 2nd ed.* Hayden: Indianapolis, Ind. 1987.

Chernicoff, Stephen. *The Macintosh Revealed, Volume Two: Programming with the Toolbox, 2nd ed.* Hayden: Indianapolis, Ind. 1987.

Chernicoff, Stephen. *The Macintosh Revealed, Volume Three: Mastering the Toolbox*. Hayden: Indianapolis, Ind. 1988.

Chernicoff, Stephen. *The Macintosh Revealed, Volume Four: Programming the Macintosh*. Hayden: Indianapolis, Ind. 1988.

Coan, James S. and Coan, Louisa. *Basic Microsoft BASIC for the Macintosh*. Howard W. Sams & Co.: Indianapolis, Ind. 1985.

Coleman, Dale and Naiman, Arthur. *The Macintosh Bible*. Goldstein & Blair. Berkeley, Calif. 1987.

Commander, Jake. *Macintosh Assembly Language Programming*. TAB Books: Blue Ridge Summit, Penn. 1985.

Crandall, Richard. *Scientific Programming with Macintosh Pascal*. John Wiley & Sons: New York, N.Y. 1986.

Cummings, Bryan and Pollack, Lawrence. *Programming the Macintosh in C*. Sybex: Alameda, Calif. 1986.

Goodman, Paul. *Advanced Macintosh Pascal*. Hayden: Indianapolis, Ind. 1987.

Haigh, Roger W. and Radford, Loren E. *Macintosh Logo*. John Wiley & Sons: New York, N.Y. 1986.

Kahler, Ted and Patterson, Dave. *A Taste of Smalltalk*. W.W. Norton & Co.: New York, N.Y. 1986.

Knaster, Scott. *How to Write Macintosh Software*. Hayden: Indianapolis, Ind. 1986.

Knaster, Scott. *Macintosh Programming Secrets*. Addison-Wesley: Reading, Mass. 1988.

Kronick, Scott. *Macintosh Pascal Illustrated*. Addison-Wesley: Reading, Mass. 1986.

Kronick, Scott. *MPW & Assembly Language Programming for the Macintosh*. Hayden: Indianapolis, Ind. 1987.

Ledgard, Henry and Singer, Andrew. *Pascal for the Macintosh*. Addison-Wesley: Reading, Mass. 1986.

Little, Gary. *Mac Assembly Language: A Guide for Programmers*. Brady: New York, N.Y. 1986.

Lu, Cary. *The Apple Macintosh Book, 2nd ed.* Microsoft Press: Redmond, Wash. 1988.

Macazine. Concepts Corporation: Austin, Tex.

MacGuide Magazine. The Delta Group: Denver, Col.

Macintosh Buyer's Guide, The. Redgate Communications Corp.: Vero Beach, Fla.

Macintosh Horizons. A.P.P.L.E. CO-OP: Renton, Wash.

MacTutor. MacTutor: Anaheim, Calif.

MacUser. Ziff-Davis Publishing Co.: New York, N.Y.

MacWorld. PCW Communications: San Francisco, Calif.

Matthews, Keith. *Assembly Language Primer for the Macintosh*. New American Library: Bergenfield, N.J. 1985.

Michel, Stephen L. *IBM PC and Macintosh Networking Featuring: TOPS and AppleShare*. Howard W. Sams & Co.: Indianapolis, Ind. 1987.

Morgan, Christopher. *Hidden Power of the Macintosh*. New American Library: Bergenfield, N.J. 1985.

Morrison, Chris and Stover, Theresa S. *PC Care Manual: Diagnosing and Maintaining Your MS-DOS, CP/M or Macintosh System*. TAB Books: Blue Ridge Summit, Penn. 1988.

Motorola. Motorola Reference Manual Package. Motorola: Phoenix, Ariz.

NibbleMac. MicroSPARC Inc.: Concord, Mass.

Niguidula, David, and Van Dam, Andries. *Pascal on the Macintosh: A Graphical Approach*. Addison-Wesley: Reading, Mass. 1987.

Poole, Lon. *Lon Poole's Mac Insights: Secrets, Shortcuts, & Solutions for the Apple Macintosh*. Microsoft Press: Redmond, Wash. 1986.

Rosenzweig, E. and Harrison., H. *Programming the 68000*. Howard W. Sams & Co.: Indianapolis, Ind. 1987.

Rosenzweig, E. and Harrison, H. *Programming the 68000 Macintosh Assembly Language*. Hayden: Indianapolis, Ind. 1986.

Schilke, T.M. and Mednieks, Z.R. *C Programming Techniques for Macintosh*. Howard W. Sams & Co.: Indianapolis, Ind. 1986.

Schmucker, Kurt. *Object-Oriented Programming for the Macintosh*. Hayden: Indianapolis, Ind. 1986.

Shafer, Dan. *Artificial Intelligence Programming on the Macintosh*. Howard W. Sams & Co.: Indianapolis, Ind. 1986.

Simonoff, Jonathan. *Introduction to Macintosh Pascal*. Howard W. Sams & Co.: Indianapolis, Ind. 1985.

Simpson, Henry. *Programming the Macintosh User Interface*. McGraw-Hill: New York, N.Y. 1986.

Smith, David E., ed. *MacTutor, Vol. 1*. MacTutor: Anaheim, Calif. 1986.

Smith, David E., ed. *MacTutor, Vol. 2*. MacTutor: Anaheim, Calif. 1986

Swan, Tom. *Programming with Macintosh Turbo Pascal*. John Wiley & Sons: New York, N.Y. 1987.

Twitty, William. *The Magic of Macintosh: Programming Graphics & Sound*. Scott, Foresman & Co.: Glenview, Ill. 1986.

Vile, Richard C., Jr. *Macintosh Programming Using MS-BASIC 2.0*. TAB Books: Blue Ridge Summit, Penn. 1985.

Waite Group, The; Stephen Prata. *Macintosh C Primer Plus*. Bantam: New York, N.Y. 1986.

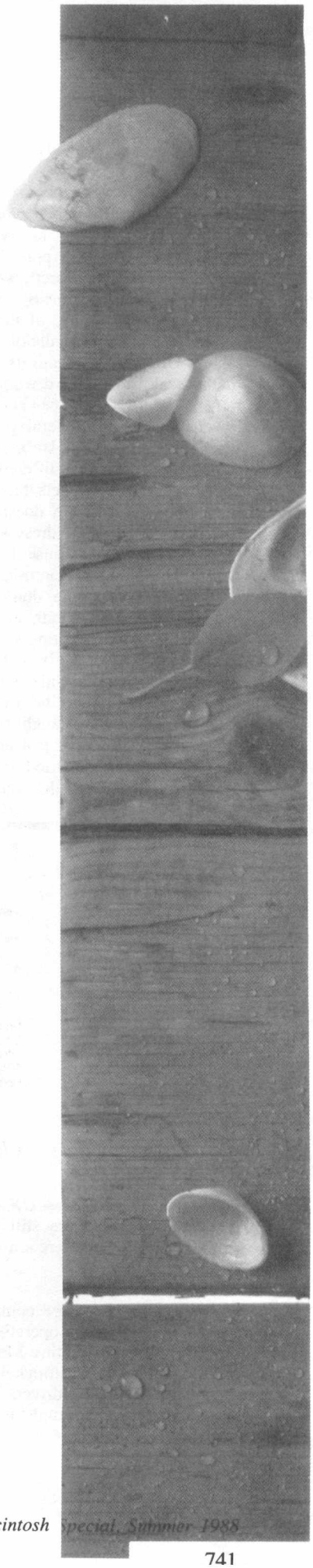
West, Joel. *Programming with Macintosh Programmer's Workshop*. Bantam: New York, N.Y. 1987.

Weston, Dan. *The Complete Book of Macintosh Assembly Language Programming, Vol. 1*. Scott Foresman & Co.: Glenview, Ill. 1986.

Weston, Dan. *The Complete Book of Macintosh Assembly Language Programming, Vol. 2*. Scott Foresman & Co: Glenview, Ill. 1987.

Willen, David C. *Macintosh Programming Techniques*. Howard W. Sams & Co.. Indianapolis, Ind. 1987.

Wilson, David. *Programming the Macintosh II*. Personal Concepts: Foster City, Calif. 1987.



APPLE'S A/UX: UNIX FOR THE REST OF US?

When is a Mac not a Mac, but still a Mac, and more

by Anthony Meadow

One of our earliest experiences with the Macintosh involved installing CP/M-68K from Digital Research. We walked through the tedious installation procedure, inserting all those 400K diskettes, to be rewarded ultimately with a text window and an *A>* prompt in an ugly font.

CP/M-68K did what it was supposed to do, but by the time we had it installed, we had all but forgotten why we had thought it would be a good idea to graft an alien and archaic operating system onto the Mac anyway.

Which is just the question some people are asking about A/UX, Apple's version of Unix for the Macintosh. A/UX wouldn't be as interesting if it only allowed you to turn off everything that makes a Mac a Mac so you could run mainframe Unix software. But as a veteran Mac programmer Tony Meadow explains in this article, how A/UX does a bit more than that. Under A/UX you can:

- Write and run Unix system software on a Macintosh
- Write and run A/UX applications that have a Macintosh interface
- Write and run applications that can be launched under either A/UX or the Macintosh operating system
- Write and run applications that run under A/UX as Unix applications (possibly with a Macintosh interface) and under the Macintosh operating system as Mac applications

This is very powerful.
Oh, and Apple solved the installation problem, too, by making the distribution medium for A/UX an 80-Mbyte hard disk. —eds

(continued from page 44)

The Rest of Us?

Why did Apple Computer work so furiously on a version of Unix tweaked for the Mac? First of all, Apple Computer's customers have been pretty vocal about wanting a Unix that runs on some flavor of a Macintosh. Many of these voices echo in the hallowed halls of Academia, where Unix first began its slow but methodical move toward world domination, or from the somewhat less hallowed halls of the federal government, where the support for Unix or the lack thereof, can make a difference in the decision about who gets those "really big" contracts. If A/UX doesn't do anything else, at least it addresses the needs of this audience, because it is a real Unix running on a Macintosh.

(What many people don't know is that A/UX, Apple's long awaited version of the Unix operating system, isn't the first version of Unix to be sold by Apple. There was a Unix available for the Lisa, but neither the Lisa nor its Unix implementation caught fire.)

Second, A/UX isn't just another vanilla Unix. Apple has tried very hard to provide all the tools that most people want in Unix. Many of these tools (Documenter's Workbench, Adobe's Transcript, the Korn shell, and so on) aren't part of the standard Unix on most other machines, but Apple has been quite generous with this well-provisioned Unix.

Why did Apple Computer work so furiously on a version of Unix tweaked for the Mac?

Third, the system administration tasks make Unix a burden when systems are used by non-experts. The A/UX tools for automating a large chunk of this work, make the Mac II and A/UX an attractive delivery vehicle for companies that want to deliver Unix software to end users, as opposed to programmers and gurus. And finally, by providing access to much of the Macintosh OS and human interface code, it's possible to develop Unix applications (or modify existing ones) that use the well-designed and well-documented Macintosh human interface. This can significantly increase the potential marketplace for some products because the Mac interface is so easy to use.

A/UX will prove attractive to many different groups of people for a variety of reasons. A/UX is attractive to the community of developers who already use Unix. Apple has simplified the licensing procedures for A/UX by providing right-to-copy licenses for 10, 25, 50, 250, and 1000 copies. The first release of A/UX:

- Provides all the tools that are expected in any version of Unix plus a lot of extra goodies such as Adobe's TranScript, AT&T's Documenter's Workbench, Sun's NFS (Networking File System), and the source code to GnuEmacs
- Reduces the need for a Unix guru by simplifying the system administration procedures and providing better tools
- Provides added value for users (and therefore developers) by providing ac-

A/UX RESOURCE GUIDE

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
Alsys Ada Compiler	Alsys Inc.	1432 Main St.	Waltham, MA 02154	617-890-0030	$3,095.00
Alsys Ada Development Tool Set	Alsys Inc.	1432 Main St.	Waltham, MA 02154	617-890-0030	$1,200.00
A/UX Network System Administration	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$18.50
A/UX Network Applications Programming	APDA	290 SW 43rd St.	Renton, WA 98055	206-251-6548	$18.00
C Compiler	Unisoft	6121 Hollis St.	Emeryville, CA 94608	415-420-6400	$495.00
FORTRAN Compiler	Unisoft	6121 Hollis St.	Emeryville, CA 94608	415-420-6400	$495.00
Informix-ESQL/Ada	Informix Software	4100 Bohannon Dr.	Menlo Park, CA 94025	415-322-4100	$1,600.00
MacFORTRAN 2.3	Absoft	1 Bond St.	Auburn Hills, MI 48057	313-853-0050	$295,$495
Pascal Compiler	Unisoft	6121 Hollis St.	Emeryville, CA 94608	415-420-6400	$495.00

A/UX - There's Always Room for Improvement

While many aspects of A/UX were well-implemented, there are still areas that could be improved. Here's a smattering of them:

- There could be more complete support of the Macintosh operating system, especially for the Printing Manager and the Layer Manager. Ultimately, support for Mac OS device drivers would be very nice too (although it probably won't be easy to do)

There could be support for AppleTalk, so that a Mac II running A/UX can exchange files, share devices (such as the LaserWriter), and so on with other Macintoshes running the Mac OS

- The X window system running under A/UX could be improved
- There could be support for developing with Object Pascal, C++ and MacApp under A/UX
- There could be support for a tape drive
- Distribution and updates should come on tapes, not on 80-Mbyte disk drives or floppy disks
- The device driver kit should be released
- There should be support for HFS formatted floppy disks (most Mac users don't use MFS disks unless they have to)
- There should be more support for color under A/UX.

cess to the Macintosh operating system and user interface code in the Macintosh II ROMs

• Allows access to the Macintosh user interface code in ROM so that programmers can develop applications with a standard Macintosh user interface (for example, A/UX can recognize when a new card has been added to the Macintosh II, locate the driver for the card, relink the kernel of the operating system to support the new device, and then reboot)

Under A/UX, all Macintosh develop-

Prices for A/UX

Description	Price Retail	Includes			
		Mac II	Monitor/Video Card	80Mb Disk (2)	Memory (3)
Monochrome Entry System	$8597.00	Y	Monochrome	Y	2MB
Color Entry System	$9346.00	Y	Color	Y	2MB
Development System	$8399.00	Y	N	Y	4MB
External Drive Upgrade Kit	$4979.00	N	N	Y (External)	4MB
Internal Drive Upgrade Kit	$4879.00	N	N	Y	4MB
External A/UX 80MB Hard Disk	$3282.00	N	N	Y (External)	0
Internal A/UX 80MB Hard Disk	$3182.00	N	N	Y	0
A/UX Manual Set (14 volumes) (1)	$649.00	N	N	N	0

Notes:
(1) Manuals must be purchased separately.
(2) The disk is an internal disk unless noted otherwise.
(3) If memory is included, then a 68851 PMMU is needed as well.

Table 1: *A/UX options*

(continued from page 47)
ment work is done in C, whether for a Mac OS application or for an A/UX application with a Mac interface. There is a set of include files and libraries (to link with) that support most of the Macintosh managers. Because some managers are not supported at all or are only partially supported, some Mac OS applications cannot be developed under A/UX. In practice, it's hard to imagine why someone would want to develop a complete Macintosh application under A/UX and not under the Mac OS, where there are better tools for some of the development stages.

There are tools for developing other categories of applications: a standard Unix application, a Unix application with a Macintosh interface, or an application that behaves as a Unix application under A/UX and as a Mac application under the Mac OS.

What do I need?

First, you need a Mac II. Because A/UX supports demand-paged virtual memory, you need to replace the MMU chip in a Macintosh II with a Motorola 68851 with a Paged Memory Management Unit (PMMU). The user address space is 512 Mbyte and the kernel address space is 4 gigabytes.

You'll also need at least 4 Mbyte of RAM to develop software in A/UX, but you can get by with as little as 2 Mbyte for running applications (although more memory is always appreciated). A/UX itself is distributed on an 80-Mbyte disk

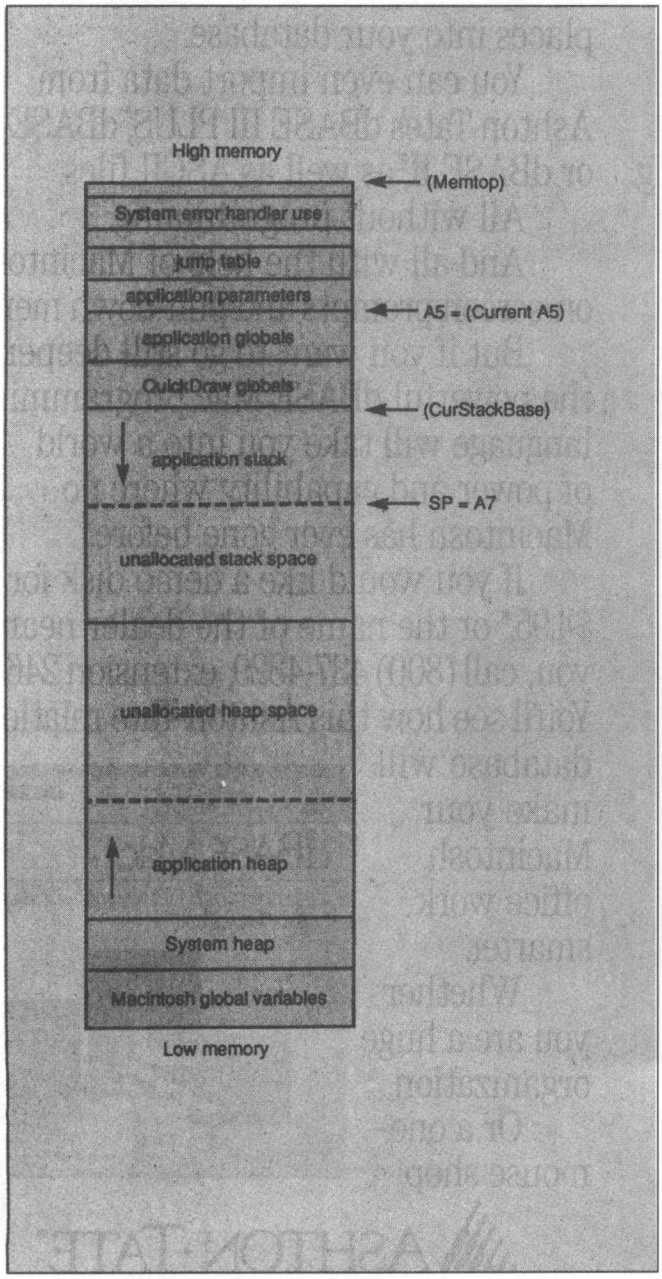


Figure 1: *Schematic memory map for a Macintosh application under Macintosh OS (on a Macintosh II)*

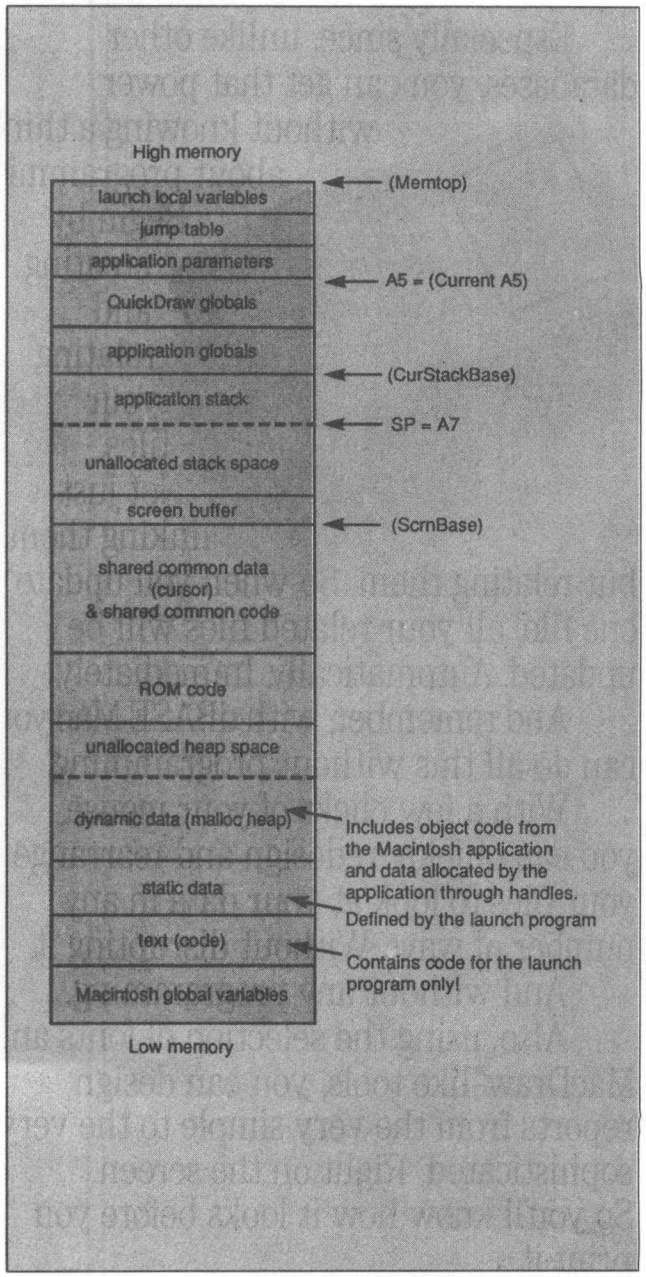


Figure 2: *Schematic memory map for a Macintosh binary file launched under A/UX*

drive with the documentation sold separately. (See Table 1, page 47, for configurations and prices.)

What Flavor of Unix?

Derived from AT&T's System V, A/UX has passed the System V Validation Suite (SVVS) and adheres to the System V Interface Definition (SVID). File locking and record locking, both mandatory and advisory, are supported in accordance with the IEEE Posix standard. While A/UX was being developed from System V, Release 2.2, AT&T finished System V, Release 3, so Apple has a little catching up to do. Apple intends to release updates to A/UX at six-month intervals.

Components of BSD Unix were spliced into A/UX because Apple wants A/UX to attract the academic community, where Berkeley Software Distribution (BSD) Versions 4.2 and 4.3 are popular. A/UX includes the BSD 4.2 signal package as well as most of the 4.2 networking code. Many commands unique to BSD are also included. In some cases, their names have been changed to avoid conflicts with System V commands.

All three of the Unix shells are included: Bourne, C, and Korn. C and Fortran77 compilers are provided as part of standard Unix. *make*, *awk*, *yacc*, *lex*, *lint*, *cb*, and all the other standard development tools are there too. The Source Code Control System (SCCS) is also included.

Many tools are provided for those who work with words. Four editors are included: *vi*, *ex*, *ed*, and *sed*. The full Documenter's Workbench includes *ditroff*, *troff*, *pic* (for creating object-oriented illustrations), *tbl* (for creating tables), and *eqn* (for creating mathematical equations). Apple also licensed TranScript from Adobe Systems, so that the LaserWriter printer can be used from A/UX as a PostScript printer. But there is no way to print from a Macintosh application running under A/UX!

The TCP/IP protocol suite is supported over Ethernet. Sun's Networking File System (NFS) is also available, including Yellow Pages. AT&T's STREAMS is also there. The first release of A/UX does not include support for AppleTalk.

The 68881 FPU can be used for floating-point calculations. Apple's Standard Apple Numeric Environment (SANE), which adheres to the IEEE 754 Standard for Binary Floating-Point Arithmetic, is also available. SANE produces more accurate results than the 68881, but it is slower because all the work is done in software.

Eschatology

Autorecovery programs simplify the system administration process by handling such problems as bad disk blocks, filesystem inconsistencies, and missing (or damaged) system files. Some routine administrative tasks must still be performed to support autorecovery, but these are fairly simple. Autorecovery doesn't do everything, though. You will still need to back up and restore your own files because autorecovery only deals with critical system files. The list of files that it is concerned about is called the configuration master list.

Many other goodies are included with A/UX. In fact, you could recover a couple of megabytes by deleting things

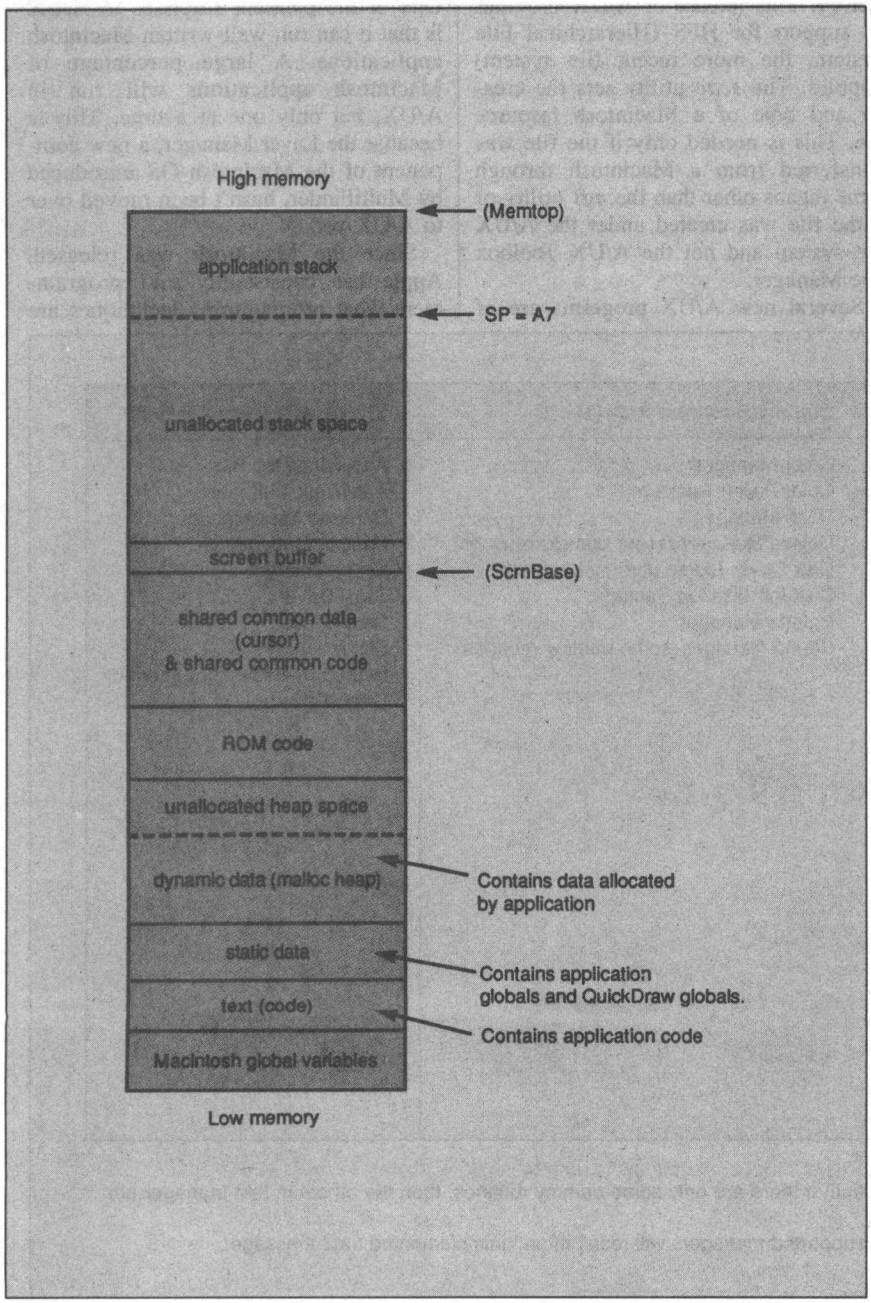


Figure 3: *Schematic memory map for an A/UX Toolbox program launched under A/UX*

(continued from page 51)
such as the complete source code for the Free Software Foundation's GnuEmacs. Kermit, the popular file transfer utility, is included, as are lots of standard Unix games (in binary form), such as adventure, fortune, trek, life, and twinkle.

Apple plans to release the source code for almost all of the drivers in A/UX (except those licensed from other companies) as part of a device driver kit.

A/UX Commands for Macintosh Applications

The *launch* command is used to start a Macintosh application. You must pass it the name of the application that you want to start and, if desired, the name(s) of one or more document files for it to open.

In order to run any Macintosh application, the *toolboxdaemon* must be running in the background. This program, which is normally started by scripts run at boot time, manages the transfer from the A/UX world to the Macintosh world and back again. The daemon is necessary because the Macintosh application environment is normally managed by the Macintosh operating system and the Finder, neither of which are present under A/UX.

Two utilities help to transfer files between A/UX and Macintosh. The *mfs* utility transfers files from Macintosh MFS disks to your current working directory. This release of A/UX provides no support for HFS (Hierarchical File System, the more recent file system) floppies. The *settc* utility sets the creator and type of a Macintosh resource file. This is needed only if the file was transferred from a Macintosh through some means other than the *mfs* utility or if the file was created under the A/UX file system and not the A/UX Toolbox File Manager.

Several new A/UX programs are of interest to anyone who plans to use a Macintosh application or an A/UX application that calls the Macintosh ROMs. A/UX also includes the *rez* and *derez* programs, which are used to compile and decompile Macintosh resources. These programs were ported from MPW (Macintosh Programmers Workshop), the native development environment from Apple for Macintosh.

A/UX and the Macintosh OS

One of the pleasant surprises in A/UX is that it can run well-written Macintosh applications. A large percentage of Macintosh applications will run in A/UX, but only one at a time. This is because the Layer Manager, a new component of the Macintosh OS introduced by MultiFinder, hasn't been moved over to A/UX yet.

Since the Macintosh was released, Apple has consistently told programmers what programming techniques are

Supported Managers	Dummy Routines Only (1)	Unsupported Managers (2)
Binary-Decimal Conversion Package	Color Manager	Apple Desktop Bus
Control Manager	Color Picker Package	AppleTalk Manager
Dialog Manager	Desk Manager	Deferred Task Manager
Event Manager, OS (partial)	Device Manager (some dummy routines)	Printing Manager
Event Manager, Toolbox	Disk Driver (some dummy routines)	Script Manager
File Manager (partial)	Disk Initialization Package	SCSI Driver
Font Manager	Palette Manager	Serial Driver
International Utilities Package	Sound Manager (some dummy routines)	Slot Manager
List Manager		Startup Manager (not needed)
Memory Manager (partial)		Video Drivers
Menu Manager		
Package Manager		
QuickDraw		
Resource Manager		
SANE Package		
Scrap Manager		
Segment Loader (partial)		
Shutdown Manager		
Standard File Package		
System Error Handler (partial)		
TextEdit		
Time Manager (partial)		
Utilities,OS (partial)		
Utilities, Toolbox		
Vertical Retrace Manager (partial)		
Window Manager		

Notes:
(1) Dummy routines will return with no action. If there are only some dummy routines, then the others in that Manager are unimplemented.
(2) Making a system call to any of the unsupported managers will result in an 'unimplemented trap' message.

Table 2: *Support for Macintosh Managers under the first release of A/UX*

good (and safe) to use. Anything explained in the five volumes of *Inside Macintosh* is usually safe and is pretty much guaranteed to apply from machine to machine. Apple has also consistently explained which techniques are not safe to use. It is particularly dangerous, for example, to use CPU specific features or instructions such as using the structure of the stack frames (which differ between the 68000 and 68020), fiddling with handles directly instead of using the appropriate system calls, and directly accessing hardware (such as the NCR 5380 SCSI or Zilog SCC chips). As it turns out, if an application was written avoiding these dangerous practices, there is a high probability that it can be run from A/UX.

One reason why properly written applications still might not run under A/UX is that not all the managers are supported under A/UX yet. In some cases, there are bugs in the Macintosh ROMs that have never appeared under the Macintosh OS. They have only appeared under A/UX because A/UX is a more demanding environment. For example, there is no hardware memory protection under the Mac OS, while A/UX uses the Motorola PMMU. Under the Mac OS, an application can directly address the Serial Communications Controller (SCC) chip, but it will be prevented from doing so by A/UX. Apple will try to make most, if not all, of the Macintosh managers available under A/UX, although it may take another release or two.

One way to develop applications that are properly written is to use MacApp. MacApp, the Expandable Macintosh Application, is an application toolkit written in Object Pascal. This language, developed by Apple Computer and Nikolaus Wirth, is a small superset of Pascal. MacApp provides all the behaviors of a standard Macintosh application. The programmer need only code those portions that are application-specific. MacApp has minimal penalties in space (10 percent) and execution time (10 percent) because the implementors of both Object Pascal and MacApp were careful in these areas. At this time, if you want to develop with MacApp, you must develop under the Macintosh operating system.

Memory Management

The Memory Manager under A/UX consists of code completely different from that used by the Macintosh OS. The A/UX Toolbox version uses the standard Unix system calls, *malloc* and *free*, to do the work of the Memory Manager. Its internal data structures are mostly different as well. A/UX master pointers are 8 bytes long rather than the 4 bytes, as in Mac OS. The flag bits, which were stored in the high-order byte of the master pointer under the Macintosh OS, are now stored in the second long word of the master pointer. You can still dereference a handle in the old way.

As future releases of A/UX are developed, there will be support for more of the Macintosh managers.

When a Macintosh application is launched, A/UX behaves as if there is 1 Mbyte of free memory. This is a compromise, because A/UX has virtual memory and the Macintosh Memory Manager was designed to manage a finite amount of physical memory. Also, A/UX has only a single heap zone—A/UX does not distinguish between the application and system heap zones.

Figure 1, page 50, which is a schematic view of the memory map of a Macintosh II, shows the traditional environment for Macintosh applications. Figure 2, page 50, a view of the virtual memory map of a Macintosh application running under A/UX, is obviously different. Any Macintosh application that makes assumptions about the order or location of various sections of memory is clearly out of luck when trying to run under A/UX.

The shared common data and code segments are managed by the *tool-*

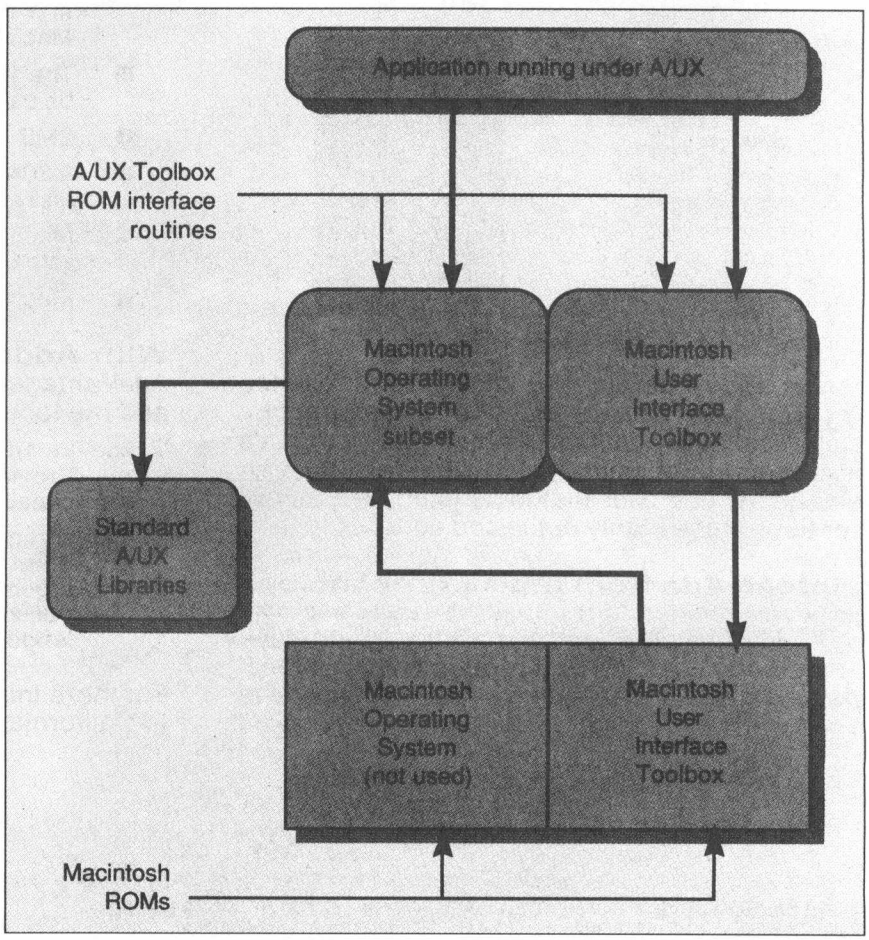


Figure 4: *Macintosh system calls to A/UX*

(continued from page 53)

boxdaemon, which runs in the background. The launch utility has to set up part of this world because it is replacing some functions of the Mac OS and the Finder.

Figure 3, page 51, shows a schematic view of the virtual memory for an A/UX program that uses the Toolbox. This environment, which does not apply to any Macintosh application, is even more different. Macintosh ROM Support A/UX supports almost all the user interface managers and many of the operating system managers. The largest number of unsupported managers is in the area of devices (SCSI Manager, AppleTalk Manager, Serial Driver, and so on). Table 2, page 52, lists all the managers supported in Release 1 of A/UX, as well as those that are supported by dummy routines and those that are not supported at all.

Dummy routines return with no error, but have no effect. Unsupported routines cause an unimplemented trap message, and the application is stopped in its tracks. As future releases of A/UX are developed, there will be support for more of the Macintosh managers.

Figure 4, page 53, illustrates how A/UX, the A/UX Toolbox and the Macintosh ROMs interact when a Macintosh system call is made. If a call is made to the operating system (rather than to the user interface), the call is handled by code in the A/UX Toolbox. This code performs the function by executing A/UX system calls to the standard A/UX libraries. The Macintosh OS code in ROM is not used at all.

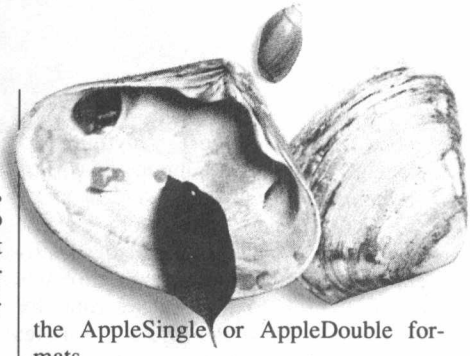
If a call is made to the user interface code, then the A/UX Toolbox first translates the parameters into a form usable by the ROM code. It does this because the ROMs were written to expect parameters that look as if they came from Pascal, rather than C. This is a tricky area because parameters that are part of a structure (or record) are not translated. The Macintosh ROM routine is then called. This routine may call other user interface toolbox routines or Macintosh OS routines.

Macintosh Files under A/UX

One area where there are some small, but tricky, differences between the two OS's is file management. There is a lot of support for Macintosh OS files under A/UX, but it isn't complete yet. A Macintosh file has two forks: data and resources. The data fork looks and behaves like any Unix file in that it is a finite sequence of bytes that can be randomly accessed. The resource fork, a concept unique to the Macintosh operating system, is normally accessed through a series of system calls.

In the Macintosh OS world, each volume appears as a single-rooted tree.

Under A/UX, a Macintosh file can be stored in one of two formats. These formats were proposed by Apple and discussed by quite a few people over Usenet, the Unix community's primary electronic network. In the AppleSingle format, there is only one A/UX file for each Macintosh file. This file contains the contents of both forks of the Macintosh file as well as all other information about the file. The latter includes a file's black and white icon, color icon, Finder information, comment (as in Get Info), and other file information, such as attributes. In the AppleDouble format, the data fork is stored as one file, and the remainder of the information is stored as another file with a % (percent) prefixed to the name of the file containing the data fork. A/UX is capable of working with Macintosh files in either



the AppleSingle or AppleDouble formats.

In the Macintosh OS world, each volume appears as a single-rooted tree. Under A/UX, all volumes appear somewhere in the file directory hierarchy, that is, the file system appears as a single-rooted tree. To Macintosh applications, all volumes appear as part of a single simulated volume whose name is / (slash). This volume cannot be mounted, unmounted, or taken on- or off-line.

Files are managed differently under the two operating systems. A/UX filenames are case-sensitive and limited to 14 characters. The Macintosh allows 64-character filenames that are not case-sensitive. Directory names are separated by / (a slash) under A/UX and by : (a colon) under the Macintosh OS. These differences will mainly affect hard-coded filenames.

Text files differ only slightly between the two operating systems. The newline character is a carriage return (0x0D) in the Macintosh operating system and a line feed (0x0A) under A/UX.

Conclusion

The Unix veterans that make up the A/UX team at Apple have delivered an applications development platform to programmers who want to tackle the academic, government, and commercial markets. Unix on the Macintosh will stimulate the Unix applications market because A/UX provides not just the Macintosh user interface but also a simplified system administration process.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

As president and co-founder of Bear River Associates Inc., P.O. Box 1900, Berkeley, CA 94701 (a software engineering department for hire), Anthony Meadow has developed Macintosh applications since 1985. Anthony is an active member of the Software Entrepreneurs' Forum. He and David Richey cofounded Bear River Institute, which develops and presents technical courses about the Macintosh.

BIBLIOGRAPHY—THE MAC II AND A/UX

Technical Documents

A/UX Network Applications Programming. APDA. Renton, Wash.

A/UX Network System Administration. APDA. Renton, Wash.

Books and Periodicals

Mac WorkStation Programmer's Guide. APDA. Renton, Wash.

Mac WorkStation Programmer's Reference. APDA. Renton, Wash.

AN INTRODUCTION TO HYPERCARD PROGRAMMING

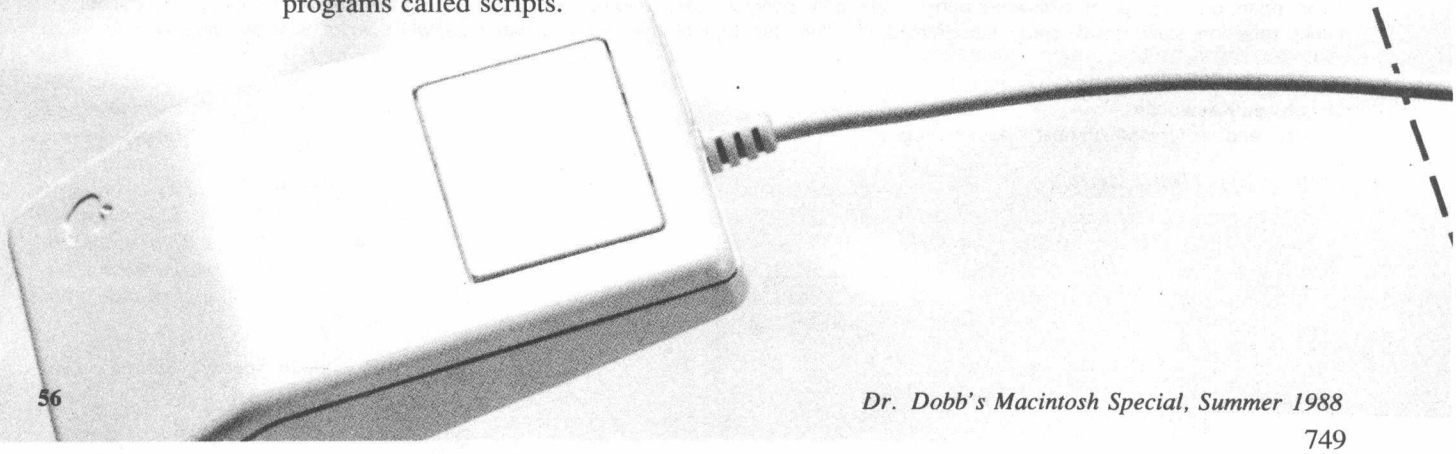
Writing XCMDs with MPW is similar to traditional Macintosh programming . . . with a few new twists.

by Dan Allen

HyperCard is a hypertext application that includes two major subsystems: a bitmap graphics editor and a programming language. Bill Atkinson, HyperCard's inventor, refers to HyperCard as a software erector set. Others see it as an information toolkit. It is not a traditional database, although it can handle large quantities of data quickly. Rather, HyperCard is a new type of application that does not fit neatly into traditional software categories.

HyperCard descended from two related applications that Bill Atkinson wrote: MacPaint[1] and QuickFile.[2] HyperCard also makes extensive use of QuickDraw, the Macintosh graphics routines found in ROM.[3] HyperCard began under the code name WildCard in late 1985. The first working developmental version was given only limited internal distribution at Apple in June 1986.

In the fall of 1986, the developers decided to include a programming language in WildCard, so Atkinson and Dan Winkler designed WildTalk. HyperCard's language, now known as HyperTalk, has a rich, English-like vocabulary that allows users to write powerful yet simple programs called scripts.



The HyperTalk language can be divided up into the following categories. This summary does not include information about the syntax of the HyperTalk vocabulary, but is simply a list of the HyperTalk lexical elements or parts of speech.

Adjectives
abbr, abbrev, abbreviated, long, short.

ChunkParts
char, character, item, line, word.

Commands
add, answer, arrowKey, ask, beep, choose, click, close, controlKey, convert, debug, delete, dial, divide, doMenu, drag, edit, enterKey, find, functionKey, get, go, help, hide, multiply, open, play, pop, print, push, put, read, reset, returnKey, set, show, sort, subtract, tabKey, type, visual, wait, write.

Constants
down, eight, empty, false, five, formFeed, four, lineFeed, nine, one, pi, quote, seven, six, space, tab, ten, three, true, two, up, zero.

Functions
abs, annuity, atan, average, charToNum, clickLoc, cmdKey, commandKey, compound, cos, date, diskSpace, exp, exp1, exp2, heapSpace, length, ln, ln1, log2, max, min, mouse, mouseClick, mouseH, mouseLoc, mouseV, number, numToChar, offset, optionKey, param, paramCount, params, random, result, round, seconds, secs, shiftKey, sin, sound, sqrt, stackSpace, tan, target, ticks, time, tool, trunc, value, version.

Messages
closeBackground, closeCard, closeField, closeStack, deleteBackground, deleteButton, deleteCard, deleteField, deleteStack, idle, mouseDown, mouseEnter, mouseLeave, mouseStillDown, mouseUp, mouseWithin, newBackground, newButton, newCard, newField, newStack, open-Background, openCard, openField, openStack, quit, resume, startup, suspend.

Months/Days
January, Jan, February, Feb, March, Mar, April, Apr, May, June, Jun, July, Jul, August, Aug, September, Sep, October, Oct, November, Nov, December, Dec, Sunday, Sun, Monday, Mon, Tuesday, Tue, Wednesday, Wed, Thursday, Thu, Friday, Fri, Saturday, Sat.

Operators
&, &&, +, –, *, /, ^, <, <=, <>, =, >, >=, and, contains, div, in, is, mod, not, or.

Ordinals
first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth, any, mid, middle, last.

Prepositions
after, before, into.

Properties.
autoHilite, blindTyping, brush, centered, cursor, dragSpeed, editBkgnd, filled, freeSize, grid, highlight, highlite, hilight, hilite, icon, id, language, lineSize, loc, location, lockMessages, lockRecent, lockScreen, lockText, multiple, multiSpace, name, numberFormat, pattern, polySides, powerKeys, rectangle, script, scroll, showLines, showName, size, style, textAlign, textArrows, textFont, textHeight, textSize, textStyle, userLevel, visible, wideMargins.

Special
all, ascending, at, background, backgrounds, barn, bkgnd, bkgnds, black, blinds, box, browse, brush, btn, bucket, button, by, can, card, cards, characters, chars, checkerboard, close, curve, cut, dateTime, descending, dialog, dissolve, door, down, effect, eraser, fast, field, file, for, forever, from, function, gray, hypercard, in, international, inverse, iris, it, lasso, left, line, me, menuBar, message, msg, modem, not, numeric, of, on, open, out, oval, paint, password, pencil, plain, poly, polygon, prev, previous, printing, rect, reg, regular, right, round, script, scroll, select, selection, slow, slowly, spray, stop, tempo, text, the, this, to, tool, until, up, venetian, very, while, white, window, wipe, with, word, zoom part2.

Structured Keywords
do, else, end, exit, global, if, next, on, pass, repeat, return, send, then.

Table 1: *HyperTalk Elements*

(continued from page 56)

HyperTalk as an Object-Oriented Language

HyperTalk follows the object-oriented programming tradition with its own flavor of objects, classes, and inheritance:

object oriented =
objects + classes + inheritance[4]

HyperCard's classes are stacks, backgrounds, cards, fields, and buttons. Users can create many new objects (which by definition are instances of these classes), but they cannot create entirely new types of classes.

Each individual object can have a script associated with it, and each HyperTalk script can contain many handlers (methods) that can send, receive, and service messages. Handlers are similar to procedures in that they can be passed parameters and called recursively, but handler definitions cannot be nested. Handlers support local and global variables.

HyperTalk's inheritance path allows messages to be dealt with by handlers at various points in the system. When a message is passed to a HyperCard object, first the script of that object is checked for a corresponding handler of the same name. If it is found, it is then interpreted: otherwise, the current cards script is checked, then the background script, then the stacks script, then the script for the home stack, and finally HyperCard itself.[5] The HyperCard commands listed in Table 1, page 58, for example, are simply messages handled by HyperCard, and thus can be redefined by the user through the inheritance path. Objects can subvert the standard hierarchy by use of the keyword send.

A Sample HyperTalk Script

The handler shown in Example 1, this page, exports all of the text from the background fields of a HyperCard stack to a text file. This handler illustrates how simple it is to write a tab- or CR-delimited ASCII file from inside of HyperCard. The method for importing is similar.

XCMDs and XFCNs

When HyperTalk is too slow or is lacking a particular function, there are a couple of trapdoors that can be used: XCMDs and XFCNs. The only difference between an XFCN and an XCMD is that an XFCN returns a result; the difference here is similar to that between a procedure and a function in Pascal. For the sake of convenience, we refer to them collectively as XCMDs.

An XCMD is compiled code. Writing XCMDs is a task for programmers, not users. With Apple's Macintosh Programmer's Workshop (MPW), programmers can write XCMDs using Assembly, C, or Pascal. For more information on MPW, see "Overview of the Macintosh Programmer's Workshop," page 20.

XCMDs are called as if they were built-in HyperTalk commands. They are invoked by HyperTalk through its inheritance mechanism. As a message ascends the hierarchy, the resource forks of certain stacks[6] are checked to see if they have an XCMD or XFCN resource of the right name that will handle the message. If the appropriate resource name is found, the resource is loaded into memory and HyperCard jumps to the start of it. After the XCMD has executed, control returns to HyperCard.

XCMDs are easily installed and moved between files with the use of *reswdit*, or better yet, with *rescopy*, a public domain stack by Apple's Steve Maller. *rescopy* imitates the Font/DA Movers interface in a HyperCard stack. (*rescopy* itself is written as an XCMD!) There are several reasons for using an XCMD:

- As an interface to drivers, in order to set up HyperCard to control external devices, such as video disk players and CD-ROMs
- As a means of accessing the Macintosh OS and ToolBox routines
- As a means of accessing some of HyperCard's internal routines
- As a means of speeding up execution of time-critical code

```
on mouseUp
  put the short name of this stack & ".tx" into
defaultName
  ask "Export text to what file?" with defaultName
  if it is empty then exit mouseUp
  put it into fileName
  open file fileName
  go to first card
  repeat for the number of cards
    repeat with i = 1 to the number of fields
      put field i into temp
      repeat
        if return is not in temp then exit repeat
        put space into char offset(return,temp) of temp
      end repeat
      write temp to file fileName
      if i _ the number of fields then
        write tab to file fileName
      else write return to file fileName
      end if
    end repeat
    go to next card
  end repeat
  close file fileName
end mouseUp
```

Example 1: *A handler to export text from the background fields of a HyperCard stack to a text file.*

XCMDs and Drivers

HyperCard, because of its outstanding user interface and its ability to be customized, is a great way to get the Macintosh to talk to other devices, such as video disk players. It is easy to control a video disk player through the built-in RS-232 ports of the Mac. XCMDs can be the glue that binds HyperCard to the video disk player.

The video disk player controller would require three pieces of code. The driver for the video disk player would be a standard Macintosh driver. These specialized pieces of code have been around since Macintish's beginnings. Many people have become proficient at writing drivers because a desk accessory is a driver. For more information on writing drivers, see *Inside Macintosh*, Volume II.

The second piece of code would be the XCMD. The XCMD would be small, and its purpose would be to simply convert HyperTalk messages to the appropriate driver calls.

The third piece of code would be the HyperTalk scripts that call the XCMD with various parameters, asking it to ask

the driver to ask the laser disk player to go to a certain frame, or to backup, for example.

Using XCMDs as an interface to traditional drivers is a simple but powerful operation. If an XCMD gets too big in size, think about writing a driver.

XCMDs and the Macintosh ToolBox

HyperTalk has a lot of functions, but if you need something it does not have, the Macintosh OS and ToolBox might have it. Indeed, Macintosh programming sometimes seems like an endless series of calls to the Mac ROMs.

Most ToolBox traps and routines can be called from XCMDs, with certain restrictions and limitations, outlined below. Keep in mind that XCMDs do have limitations. They are not allowed to do everything that an application is allowed to do because they are guests in HyperCard's heap. XCMDs are more like desk accessories than an application in this regard.[7] Listing One on page 66 contains a sample HyperCard XFCN, which returns the contents of memory.

Guidelines for writings XCMDs are as follows:

- Do not initialize the various Macintosh managers by calling their *init* traps: that is, do not call *InitGraf*, *InitFonts*, *InitWindows*, and so on.
- Do not rely upon having lots of RAM available for your code. There is some extra space in HyperCard's heap, but if HyperCard is running in 750K under MultiFinder, for example, no XCMD should exceed 32K.
- Do not use register A5 of the 68xxx processor. The value in A5 is HyperCard's and points to HyperCard's global data, jump table, and other things that constitute an A5 World. XCMDs do not currently have their own A5 World.
- No A5 World means no global data for XCMDs.
- No global data means no use of string literals with MPW C, since MPW C makes string literals into global data. (Alternative: Use STR resources or put the strings in a short assembly glue file.)
- No A5 World means no jump table. No jump table means no code segments. No code segments means a 32K limit on code size for 68000 based machines. (The 68020 supports longer branches.)

Writing XCMDs is a task for programmers, not users.

- XCMDs can, however, allocate small chunks of memory by standard NewHandle (and if you really must do it, *NewPointer*) calls.
- If your code allocated some memory in the heap, your code should also deallocate the memory.
- If an XCMD allocates a handle to save state information between invocations of the XCMD, then you must also use HyperTalk to store the handle somewhere in the current stack, perhaps in a hidden field. You will need to convert the handle from a *LongInt* to a string, as everything is treated as a string on the HyperTalk end of things.
- Since HyperTalk jumps blindly to the start of an XCMD piece of code, it is important that the main routine actually end up at the start of the XCMD: The link order is important.

XCMDs and HyperTalk Callback Routines

There are many useful internal routines that HyperTalk also allows XCMDs to call. HyperTalk provides both conversion routines as well as stackaccess routines.

The conversion routines provide support for converting various styles of strings and numbers into each other. For example, there are routines to convert C-style strings (zero terminated) back and forth to Pascal-style strings (length byte followed by string). There are also routines to convert strings to and from integers. Listing Two on page 66 contains a sample HyperCard XCMD to highlight the screen.

The stackaccess routines allow an XCMD to call back into HyperTalk and send messages to the current card or to HyperCard itself. In addition, values of variables can be recalled and stored into, and contents of fields can be examined. The bulk of the code in the glue files is there to support these callback routines.

XCMDs and Speeding up HyperTalk

HyperTalk is versatile, partly because it is an interpreted language. However, because it is interpreted, it is also sometimes slow, especially when doing anything repetitive. For this reason, it is often wise to rewrite some operations as XCMDs, to benefit from the speed of compiled native 68xxx code.

HYPERCARD RESOURCE GUIDE

Product	Vendor	Street Address	City, State, Zip	Phone Number	Retail Price
Dan Shafer's Script Expert	HyperPress Publishing	P.O. Box 8243	Foster City, CA 94404	415-345-4620	$79.95
Hyperbug (on AppleLink network)	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	
HyperCard	Apple Computer	20525 Mariani Ave.	Cupertino, CA 95014	408-996-1010	$49.00
HyperCard Construction Kit	Telesis SoftwareSystems	4525 Northpark Dr., Ste. 101	Colorado Springs, CO 80918	719-593-7377	*$199.95*
HyperDA	Symmetry Corp.	761 E. University Dr.	Mesa, AZ 85203	602-844-2199	$69.95
Hyper Lab	ComputerWare Learning Center	490 California Ave.	Palo Alto, CA 94306	415-323-7951	
HyperSpell	HyperPress Publishing	P.O. Box 8243	Foster City, CA 94404	415-345-4620	$79.95
HyperTutor	Teligraphics	936 Sir Francis Drake Blvd., #R	Kentfield, CA 94904	415-454-7519	$49.95
Icon Factory	HyperPress Publishing	P.O. Box 8243	Foster City, CA 94404	415-345-4620	$49.95
Inside Out	Shana Enterprises	650 20th Ave. #105ATC	Edmonton, Alberta, Canada T6V 1G1	403-438-6548	$395.00
New Introduction to HyperCard	ComputerWare Learning Center	490 California Ave.	Palo Alto, CA 94306	415-323-7951	
Script Expert	HyperPress Publishing	P.O. Box 8243	Foster City, CA 94404	451-345-4620	
StackPak	EduComp	742 Genevieve, Ste. D, Dept. H	Solana Beach, CA 92075	800-843-9497	
Visual Interactive Programming (VIP)	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$124.95
VIP to Lightspeed C Translator	Mainstay	5311-B Derry Ave.	Agoura Hills, CA 91301	818-991-6540	$124.95
1stTEAM with HyperAccess	1stDESK Systems	7 Industrial Park Rd.	Medway, MA 02053	617-533-2203	

Documentation

HyperCard ships with an excellent introductory manual that, unfortunately, does not include a description of HyperTalk. However, several good sources of information about using HyperTalk are available. The Help stack that ships with HyperCard contains a great deal of information about the language, and is an excellent resource.

The most accurate single source about HyperTalk is the reference manual, *HyperCard Script Language Guide*, written by Alan Spragens of Apple Computer. A good tutorial (with pretty good reference material) for HyperTalk has become an overnight phenomenon: Danny Goodman's *The Complete HyperCard Handbook*. This book, published by Bantam, is currently a bestseller. There is a rash of other books on the market, but most of them are highly inaccurate. Reader be warned!

The official documentation on XCMDs is available from the Apple Programmer's and Developer's Association (APDA). *The HyperCard Script Language Guide* has some technical material on writing XCMDs, but the HyperCard Developer's Toolkit contains the bulk of the information, in this case as example XCMDs in C and Pascal for MPW. For more information, contact: Apple Programmer's and Developer's Assoc., 290 SW 43rd Street, Renton, WA 98055; 206-251-6548.

As of January 1, 1988, the following HyperCard related products are sold through APDA. All software is shipped on 800K HFS disks. *HyperCard Script Language Guide* (200 pages), $18.50. *HyperCard Developer's Toolkit* (One disk, 20 pages), $10.00. The disk includes the header and glue files shown in Listings Three (page 68) and Four (page 70).

Notes

1. Originally packaged with every Macintosh sold, the original versions of MacPaint (1.0 through 1.5) were written 75 percent in Pascal and 25 percent in assembly language on a Lisa, using the Lisa Monitor, a smaller, faster precursor of the Lisa Workshop. The latest version of MacPaint (2.0) is a major revision by David Ramsey using MPW Pascal. MacPaint is now available through Apple's spin-off software company, Claris.
2. QuickFile is a small publicdomain file manager that Bill Atkinson wrote in 1985. It uses a fast in-memory text search with a fixed-sized Rolodex card. Its search routine is completely different from that used in HyperCard. QuickFile was originally called Rolodex, but the name was changed due to copyright problems.
3. Bill Atkinson originally wrote QuickDraw for the Lisa computer. When Apple decided to include QuickDraw on the Macintosh, Bill became an honorary Mac team member. QuickDraw was first written in Pascal, then rewritten many times, ending up finally as 24K of 68000 assembly code. Bill further enhanced QuickDraw for the Mac Plus by unwinding some of the loops, thus increasing the speed. However, Bill was not involved with Color QuickDraw, which is found on the Mac II, because he was involved with HyperCard at the time. Ernie Beernink and Dave Fung did Color QuickDraw.
4. This equation is from Peter Wegner's OOPSLA '87 paper, "Dimensions of Object-Based Language Design." For more definitional information about object-oriented programming, see "Overview of the Macintosh Programmer's Workshop," page xx.
5. Actually the inheritance path is slightly more complicated when the currently executing script goes to a different card or stack during its execution. The current script card or stack is searched as well as the current (visible) card or stack.
6. The search order for XCMDs is: (a) the stack that contains the running script, (b) the current stack (if different from a) (c) the Home stack, (d) HyperCard itself.
7. Desk accessories were originally meant to be a maximum of 8K. Today there are desk accessories over 100K! XCMDs of that size may work, but the system was really only designed for XCMDs of about 32K.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 68.)

Vote for you favorite feature/article.
Circle Reader Service **No. 2**

Dan Allen is a Software Explorer for Apple Computer where he has worked on several projects including MacApp, HFS, MacPlus, MacsBug, and MPW. He is currently working with Bill Atkinson on HyperCard. He can be reached at Apple Computer Inc., 20525 Mariani Ave., MS: 27E, Cupertino, CA 95014.

BIBLIOGRAPHY—HYPERCARD AND HYPERTALK

Technical Documents

HyperCard Developer's Toolkit Version 1.0. APDA. Renton, Wash.
HyperCard Script Language Guide. APDA. Renton, Wash.
HyperCard Technical Reference. APDA. Renton, Wash. Final edition is to be published in early 1988 by Addison-Wesley: Reading, Mass.

Books and Periodicals

Gerwitz, David Allen. *HyperTalk Pocket Reference*. Softpress Publishing: Foster City, Calif. 1988.
Goodman, Danny. *The Complete HyperCard Hand Book*. Bantam: New York, N.Y. 1987.
Harvey, Greg. *Understanding HyperCard*. Sybex: Alameda, Calif. 1988.
HyperAge. HyperAge. Riverdale, N.Y.
HyperLink. HyperLink Magazine. Eugene, Ore.
HyperNews. The Delta Group. Denver, Col.
Kaehler, Carol. *HyperCard Power: Techniques and Scripts*. Addison-Wesley: Reading, Mass. 1988.
Open Stack Letter, The. Walking Shadow Press: Saratoga, Calif.
Sanders, William B. *HyperCard Made Easy*. Scott, Foresman & Co.: Glenview, Ill. 1988.
Shafer, Dan. *HyperTalk Programming*. Hayden Books: Indianapolis, Ind. 1988.
Stoddard, Jeff. *HyperCard Scripting*. Walking Shadow Press: Saratoga, Calif. 1987.
Windoid. AHUG (Apple HyperCard User's Group) c/o Dave Leffler, Mail Stop 27AQ. Apple Computer: Cupertino, Calif. 1988.

Listing One *(Text begins on page 56)*

```
/*
 *        Peek.c        -            A sample HyperCard XFCN to return the contents of memory
 *                                   -            Copyright Apple Computer, Inc.  1987,1988.
 *                                   -            All Rights Reserved.
 *
 *        Build instructions:
 *
 *        C Peek.c -o Peek.c.o
 *        Link Peek.c.o -sg CPeek -rt XFCN=6 -o StackName
 *
 */

#define __SEG__ CPeek   /* Segment name must be the same as command name */

#include <HyperXCmd.h>  /* HT interface and #includes Types.h, Memory.h */

#define PEEKBYTE(address)            *((char *) address)
#define PEEKWORD(address)            *((short *) address)
#define PEEKLONG(address)            *((long *) address)

/*
 *        Your routine MUST be the first code that is generated in the file, as
 *        HyperTalk simply JSRs to the start of the XFCN segment in memory.
 *  Therefore the XCmdGlue.c file must be included after the main routine,
 *        being CPeek in this sample XFCN.  Also note that XFCNs do not currently
 *        support their own A5 World, thus NO GLOBAL VARIABLES are allowed.
 *  If the link fails then that means the C compiler generated A5-relative
 *        code.  (This happens if you try to use the C libraries or use strings
 *        in the code.  Use a STR resource instead.)
 *
 */

pascal void CPeek(paramPtr)
        XCmdBlockPtr  paramPtr;
{
        char          str[255];
        short         argc;
        long          peekAddr, peekSize, peekVal;
        Handle        argv1, argv2;

        argc  = paramPtr->paramCount;
        argv1 = paramPtr->params[0];
        argv2 = paramPtr->params[1];

        ZeroToPas(paramPtr, *argv1, str);                    /* CtoP string */
        peekAddr = StrToNum(paramPtr, str);                  /* get address */

        if (argc == 2) {
                ZeroToPas(paramPtr, *argv2, str);            /* CtoP string */
                peekSize = StrToNum(paramPtr, str);          /* get size */
        }
        else
                peekSize = 1;

        switch(peekSize) {
                case 1:                  peekVal = PEEKBYTE(peekAddr); break;
                case 2:      peekVal = PEEKWORD(peekAddr); break;
                case 4:      peekVal = PEEKLONG(peekAddr); break;
                default:     peekVal = 0;
        }
        NumToStr(paramPtr, peekVal, str);

        /* XFCN: make sure to return a result, the only change from an XCMD */
        paramPtr->returnValue = PasToZero(paramPtr, str);
}

#include <XCmdGlue.c>
```

End Listing One

Listing Two

```
/*
 *        Flash.c       -            A sample HyperCard XCMD to hilite the screen
 *                                   -            Copyright Apple Computer, Inc.  1987-1988.
 *                                   -            All Rights Reserved.
 *
 *        Build instructions:
 *
 *        C  Flash.c -o Flash.c.o
 *        Link Flash.c.o -sg CFlash -rt XCMD=5 -o StackName
 *
 */

#define __SEG__ CFlash  /* Segment name must be the same as command name */
```

```
#include <HyperXCmd.h>  /* HT interface and #includes Types.h, Memory.h */
#include <QuickDraw.h>

/*
 *        Your routine MUST be the first code that is generated in the file, as
 *        HyperTalk simply JSRs to the start of the XCMD segment in memory.
 *  Therefore the XCmdGlue.c file must be included after the main routine,
 *        being CFlash in this sample XCMD.  Also note that XCMDs do not currently
 *        support their own A5 World, thus NO GLOBAL VARIABLES are allowed.
 *  If the link fails then that means the C compiler generated A5-relative
 *        code.  (This happens if you try to use the C libraries or use strings
 *        in the code.  Use a STR resource instead.)
 *
 */

pascal void CFlash(paramPtr)
          XCmdBlockPtr  paramPtr;
{
          short         flashCount,again;
          GrafPtr       port;
          Str255        str;

          ZeroToPas(paramPtr,*(paramPtr->params[0]),&str); /*    get flash count */
          flashCount = StrToNum(paramPtr,&str);                          /* convert to num */
          if (paramPtr->paramCount != 1) flashCount = 3;          /*          default if no param */
          if (flashCount < 1) flashCount = 3;                                    /* must be positive */
          GetPort(&port);
          for (again = 1; again <= flashCount; again++) {
                    InvertRect(&port->portRect);
                    InvertRect(&port->portRect);
          }
}

#include <XCmdGlue.c>   /* C routines for HyperCard callbacks */
```

End Listing Two

Listing Three

```
/*
 *        HyperXCmd.h   - Interfaces for HyperTalk callback routines
 *                                                - Copyright Apple Computer, Inc. 1987,1988.
 *                                                - All Rights Reserved.
 *
 *        #include this file before your program.
 *        #include "XCmdGlue.c" after your code.
 *
 */

#include <Types.h>
#include <Memory.h>

pascal void Debugger() extern 0xA9FF;

typedef struct XCmdBlock {
          short         paramCount;
    Handle  params[16];
    Handle  returnValue;
    Boolean             passFlag;

    void   (*entryPoint)();    /* to call back to HyperCard */
    short  request;
    short  result;
    long   inArgs[8];
    long   outArgs[4];
} XCmdBlock, *XCmdBlockPtr;

typedef struct Str31 {
          char          guts[32];
} Str31, *Str31Ptr;


/* result codes */
#define   xresSucc                  0
#define   xresFail                  1
#define   xresNotImp                2

/* request codes */
#define   xreqSendCardMessage                   1
#define   xreqEvalExpr                          2
#define   xreqStringLength                      3
#define   xreqStringMatch                       4
#define xreqSendHCMessage           5
#define   xreqZeroBytes             6
#define   xreqPasToZero                         7
#define   xreqZeroToPas                         8
#define   xreqStrToLong                         9
#define   xreqStrToNum                          10
#define   xreqStrToBool                         11
#define   xreqStrToExt                          12
#define   xreqLongToStr                         13
#define   xreqNumToStr                          14
#define   xreqNumToHex                          15
#define   xreqBoolToStr                         16
#define   xreqExtToStr                          17
#define   xreqGetGlobal                         18
#define   xreqSetGlobal                         19
#define   xreqGetFieldByName                    20
#define   xreqGetFieldByNum                     21
#define   xreqGetFieldByID                      22
#define   xreqSetFieldByName                    23
#define   xreqSetFieldByNum                     24
#define   xreqSetFieldByID                      25
#define   xreqStringEqual           26
#define   xreqReturnToPas           27
```

```
#define    xreqScanToReturn            28
#define    xreqScanToZero              39   /* was suppose to be 29!  Oops! */

/* Forward definitions of glue routines.  Main program
        must include XCmdGlue.c after its routines.  */

pascal void SendCardMessage(paramPtr,msg)
        XCmdBlockPtr  paramPtr;     StringPtr     msg;          extern;
pascal Handle EvalExpr(paramPtr,expr)
        XCmdBlockPtr  paramPtr;     StringPtr     expr;         extern;
pascal long StringLength(paramPtr,strPtr)
        XCmdBlockPtr  paramPtr;     StringPtr     strPtr;       extern;
pascal Ptr StringMatch(paramPtr,pattern,target)
        XCmdBlockPtr  paramPtr;     StringPtr     pattern;
        Ptr           target;       extern;
pascal void SendHCMessage(paramPtr,msg)
        XCmdBlockPtr  paramPtr;     StringPtr     msg;          extern;
pascal void ZeroBytes(paramPtr,dstPtr,longCount)
        XCmdBlockPtr  paramPtr;     Ptr           dstPtr;
        long          longCount;    extern;
pascal Handle PasToZero(paramPtr,pasStr)
        XCmdBlockPtr  paramPtr;     StringPtr     pasStr;       extern;
pascal void ZeroToPas(paramPtr,zeroStr,pasStr)
        XCmdBlockPtr  paramPtr;     char          *zeroStr;
        StringPtr     pasStr;       extern;
pascal long StrToLong(paramPtr,strPtr)
        XCmdBlockPtr  paramPtr;     Str31 *       strPtr;       extern;
pascal long StrToNum(paramPtr,str)
        XCmdBlockPtr  paramPtr;     Str31 *       str;          extern;
pascal Boolean StrToBool(paramPtr,str)
        XCmdBlockPtr  paramPtr;     Str31 *       str;          extern;
pascal void StrToExt(paramPtr,str,myext)
        XCmdBlockPtr  paramPtr;     Str31 *       str;
        extended *    myext;        extern;
pascal void LongToStr(paramPtr,posNum,mystr)
        XCmdBlockPtr  paramPtr;     long          posNum;
        Str31 *       mystr;        extern;
pascal void NumToStr(paramPtr,num,mystr)
        XCmdBlockPtr  paramPtr;     long          num;
        Str31 *       mystr;        extern;
pascal void NumToHex(paramPtr,num,nDigits,mystr)
        XCmdBlockPtr  paramPtr;     long          num;
        short         nDigits;                                  Str31 *       mystr;        extern;
pascal void BoolToStr(paramPtr,bool,mystr)
        XCmdBlockPtr  paramPtr;     Boolean       bool;
        Str31 *       mystr;        extern;
pascal void ExtToStr(paramPtr,myext,mystr)
        XCmdBlockPtr  paramPtr;     extended *    myext;
        Str31 *       mystr;        extern;
pascal Handle GetGlobal(paramPtr,globName)
        XCmdBlockPtr  paramPtr;     StringPtr     globName;     extern;
pascal void SetGlobal(paramPtr,globName,globValue)
        XCmdBlockPtr  paramPtr;     StringPtr     globName;
        Handle        globValue;    extern;
pascal Handle GetFieldByName(paramPtr,cardFieldFlag,fieldName)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        StringPtr     fieldName;    extern;
pascal Handle GetFieldByNum(paramPtr,cardFieldFlag,fieldNum)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        short         fieldNum;     extern;
pascal Handle GetFieldByID(paramPtr,cardFieldFlag,fieldID)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        short         fieldID;      extern;
pascal void SetFieldByName(paramPtr,cardFieldFlag,fieldName,fieldVal)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        StringPtr     fieldName;    Handle        fieldVal;     extern;
pascal void SetFieldByNum(paramPtr,cardFieldFlag,fieldNum,fieldVal)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        short         fieldNum;                                 Handle        fieldVal;     extern;
pascal void SetFieldByID(paramPtr,cardFieldFlag,fieldID,fieldVal)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        short         fieldID;                                  Handle        fieldVal;     extern;
pascal Boolean StringEqual(paramPtr,str1,str2)
        XCmdBlockPtr  paramPtr;     Str31 *       str1;
        Str31 *       str2;         extern;
pascal void ReturnToPas(paramPtr,zeroStr,pasStr)
        XCmdBlockPtr  paramPtr;     Ptr           zeroStr;
        StringPtr     pasStr;       extern;
pascal void ScanToReturn(paramPtr,scanHndl)
```

(continued on next page)

Listing Three *(Listing continued, text begins on page 56)*

```
          XCmdBlockPtr  paramPtr;     Ptr *         scanHndl;     extern;
pascal void ScanToZero(paramPtr,scanHndl)
          XCmdBlockPtr  paramPtr;     Ptr *         scanHndl;     extern;
```

End Listing Three

Listing Four

```
/*
 *        XCmdGlue.c    - Implementation of HyperTalk callback routines
 *                                              - Copyright Apple Computer, Inc. 1987,1988.
 *                                              - All Rights Reserved.
 *
 *        #include "HyperXCmd.h" before your program.
 *        #include this file after your code.
 *
 */

pascal void SendCardMessage(paramPtr,msg)
          XCmdBlockPtr  paramPtr;     StringPtr     msg;
          /* Send a HyperCard message (a command with arguments) to the current card.
             msg is a pointer to a Pascal format string.  */
{
          paramPtr->inArgs[0] = (long)msg;
          paramPtr->request = xreqSendCardMessage;
    paramPtr->entryPoint();
}


pascal Handle EvalExpr(paramPtr,expr)
          XCmdBlockPtr  paramPtr;     StringPtr     expr;
          /* Evaluate a HyperCard expression and return the answer.  The answer is
             a handle to a zero-terminated string. */
{
          paramPtr->inArgs[0] = (long)expr;
          paramPtr->request = xreqEvalExpr;
    paramPtr->entryPoint();
          return (Handle)paramPtr->outArgs[0];
}


pascal long StringLength(paramPtr,strPtr)
          XCmdBlockPtr  paramPtr;     StringPtr     strPtr;
/* Count the characters from where strPtr points until the next zero byte.
   Does not count the zero itself.  strPtr must be a zero-terminated string. */
{
          paramPtr->inArgs[0] = (long)strPtr;
          paramPtr->request = xreqStringLength;
    paramPtr->entryPoint();
          return (long)paramPtr->outArgs[0];
}


pascal Ptr StringMatch(paramPtr,pattern,target)
          XCmdBlockPtr  paramPtr;     StringPtr     pattern;      Ptr           target;
/* Perform case-insensitive match looking for pattern anywhere in
   target, returning a pointer to first character of the first match,
   in target or NIL if no match found.  pattern is a Pascal string,
   and target is a zero-terminated string. */
{
          paramPtr->inArgs[0] = (long)pattern;
          paramPtr->inArgs[1] = (long)target;
          paramPtr->request = xreqStringMatch;
    paramPtr->entryPoint();
          return (Ptr)paramPtr->outArgs[0];
}


pascal void SendHCMessage(paramPtr,msg)
          XCmdBlockPtr  paramPtr;     StringPtr     msg;
          /* Send a HyperCard message (a command with arguments) to HyperCard.
             msg is a pointer to a Pascal format string.  */
{
          paramPtr->inArgs[0] = (long)msg;
          paramPtr->request = xreqSendHCMessage;
    paramPtr->entryPoint();
}


pascal void ZeroBytes(paramPtr,dstPtr,longCount)
```

```
        XCmdBlockPtr  paramPtr;     Ptr          dstPtr;       long         longCount;
/* Write zeros into memory starting at destPtr and going for longCount
   number of bytes. */
{
        paramPtr->inArgs[0] = (long)dstPtr;
        paramPtr->inArgs[1] = longCount;
        paramPtr->request = xreqZeroBytes;
    paramPtr->entryPoint();
}


pascal Handle PasToZero(paramPtr,pasStr)
        XCmdBlockPtr  paramPtr;     StringPtr     pasStr;
/* Convert a Pascal string to a zero-terminated string.  Returns a handle
   to a new zero-terminated string.  The caller must dispose the handle.
   You'll need to do this for any result or argument you send from
   your XCMD to HyperTalk. */
{
        paramPtr->inArgs[0] = (long)pasStr;
        paramPtr->request = xreqPasToZero;
    paramPtr->entryPoint();
        return (Handle)paramPtr->outArgs[0];
}


pascal void ZeroToPas(paramPtr,zeroStr,pasStr)
        XCmdBlockPtr  paramPtr;     char         *zeroStr;     StringPtr     pasStr;
/* Fill the Pascal string with the contents of the zero-terminated
   string.  You create the Pascal string and pass it in as a VAR
   parameter.  Useful for converting the arguments of any XCMD to
   Pascal strings. */
{
        paramPtr->inArgs[0] = (long)zeroStr;
        paramPtr->inArgs[1] = (long)pasStr;
        paramPtr->request = xreqZeroToPas;
    paramPtr->entryPoint();
}


pascal long StrToLong(paramPtr,strPtr)
        XCmdBlockPtr  paramPtr;     Str31 *       strPtr;
/* Convert a string of ASCII decimal digits to an unsigned long integer. */
{
        paramPtr->inArgs[0] = (long)strPtr;
        paramPtr->request = xreqStrToLong;
    paramPtr->entryPoint();
        return (long)paramPtr->outArgs[0];
}


pascal long StrToNum(paramPtr,str)
        XCmdBlockPtr  paramPtr;     Str31 *       str;
/* Convert a string of ASCII decimal digits to a signed long integer.
   Negative sign is allowed. */
{
        paramPtr->inArgs[0] = (long)str;
        paramPtr->request = xreqStrToNum;
    paramPtr->entryPoint();
        return paramPtr->outArgs[0];
}


pascal Boolean StrToBool(paramPtr,str)
        XCmdBlockPtr  paramPtr;     Str31 *       str;
/* Convert the Pascal strings 'true' and 'false' to booleans. */
{
        paramPtr->inArgs[0] = (long)str;
        paramPtr->request = xreqStrToBool;
    paramPtr->entryPoint();
        return (Boolean)paramPtr->outArgs[0];
}


pascal void StrToExt(paramPtr,str,myext)
        XCmdBlockPtr  paramPtr;     Str31 *       str;          extended *    myext;
 /* Convert a string of ASCII decimal digits to an extended long integer.
        Instead of returning a new extended, as Pascal does, it expects you
        to create myext and pass it in to be filled. */
{
        paramPtr->inArgs[0] = (long)str;
```

(continued on next page)

Listing Four *(Listing continued, text begins on page 56)*

```
          paramPtr->inArgs[1] = (long)myext;
          paramPtr->request = xreqStrToExt;
    paramPtr->entryPoint();
}
pascal void LongToStr(paramPtr,posNum,mystr)
          XCmdBlockPtr  paramPtr;     long          posNum;       Str31 *       mystr;
 /* Convert an unsigned long integer to a Pascal string.  Instead of
    returning a new string, as Pascal does, it expects you to
          create mystr and pass it in to be filled. */
{
          paramPtr->inArgs[0] = (long)posNum;
          paramPtr->inArgs[1] = (long)mystr;
          paramPtr->request = xreqLongToStr;
    paramPtr->entryPoint();
}


pascal void NumToStr(paramPtr,num,mystr)
          XCmdBlockPtr  paramPtr;     long          num;          Str31 *       mystr;
 /* Convert a signed long integer to a Pascal string.  Instead of
    returning a new string, as Pascal does, it expects you to
          create mystr and pass it in to be filled. */
{
          paramPtr->inArgs[0] = num;
          paramPtr->inArgs[1] = (long)mystr;
          paramPtr->request = xreqNumToStr;
    paramPtr->entryPoint();
}


pascal void NumToHex(paramPtr,num,nDigits,mystr)
          XCmdBlockPtr  paramPtr;     long          num;
          short         nDigits;                                Str31 *       mystr;
/* Convert an unsigned long integer to a hexadecimal number and put it
   into a Pascal string.  Instead of returning a new string, as
   Pascal does, it expects you to create mystr and pass it in to be filled. */
{
          paramPtr->inArgs[0] = num;
          paramPtr->inArgs[1] = nDigits;
          paramPtr->inArgs[2] = (long)mystr;
          paramPtr->request = xreqNumToHex;
    paramPtr->entryPoint();
}


pascal void BoolToStr(paramPtr,bool,mystr)
          XCmdBlockPtr  paramPtr;     Boolean       bool;         Str31 *       mystr;
 /* Convert a boolean to 'true' or 'false'.  Instead of returning
    a new string, as Pascal does, it expects you to create mystr
          and pass it in to be filled. */
{
          paramPtr->inArgs[0] = (long)bool;
          paramPtr->inArgs[1] = (long)mystr;
          paramPtr->request = xreqBoolToStr;
    paramPtr->entryPoint();
}


pascal void ExtToStr(paramPtr,myext,mystr)
          XCmdBlockPtr  paramPtr;     extended *    myext;        Str31 *       mystr;
 /* Convert an extended long integer to decimal digits in a string.
    Instead of returning a new string, as Pascal does, it expects
          you to create mystr and pass it in to be filled. */
{
          paramPtr->inArgs[0] = (long)myext;
          paramPtr->inArgs[1] = (long)mystr;
          paramPtr->request = xreqExtToStr;
    paramPtr->entryPoint();
}


pascal Handle GetGlobal(paramPtr,globName)
          XCmdBlockPtr  paramPtr;     StringPtr     globName;
/* Return a handle to a zero-terminated string containing the value of
   the specified HyperTalk global variable. */
{
          paramPtr->inArgs[0] = (long)globName;
          paramPtr->request = xreqGetGlobal;
```

```
    paramPtr->entryPoint();
          return (Handle)paramPtr->outArgs[0];
}
pascal void SetGlobal(paramPtr,globName,globValue)
          XCmdBlockPtr  paramPtr;     StringPtr     globName;     Handle        globValue;
/* Set the value of the specified HyperTalk global variable to be
   the zero-terminated string in globValue.  The contents of the
   Handle are copied, so you must still dispose it afterwards.  */
{
          paramPtr->inArgs[0] = (long)globName;
          paramPtr->inArgs[1] = (long)globValue;
          paramPtr->request = xreqSetGlobal;
    paramPtr->entryPoint();
}


pascal Handle GetFieldByName(paramPtr,cardFieldFlag,fieldName)
          XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
          StringPtr     fieldName;
/* Return a handle to a zero-terminated string containing the value of
   field fieldName on the current card.  You must dispose the handle. */
{
          paramPtr->inArgs[0] = (long)cardFieldFlag;
          paramPtr->inArgs[1] = (long)fieldName;
          paramPtr->request = xreqGetFieldByName;
    paramPtr->entryPoint();
          return (Handle)paramPtr->outArgs[0];
}


pascal Handle GetFieldByNum(paramPtr,cardFieldFlag,fieldNum)
          XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
          short         fieldNum;
/* Return a handle to a zero-terminated string containing the value of
   field fieldNum on the current card.  You must dispose the handle. */
{
          paramPtr->inArgs[0] = (long)cardFieldFlag;
          paramPtr->inArgs[1] = fieldNum;
          paramPtr->request = xreqGetFieldByNum;
    paramPtr->entryPoint();
          return (Handle)paramPtr->outArgs[0];
}


pascal Handle GetFieldByID(paramPtr,cardFieldFlag,fieldID)
          XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
          short         fieldID;
/* Return a handle to a zero-terminated string containing the value of
   the field whise ID is fieldID.  You must dispose the handle. */
{
          paramPtr->inArgs[0] = (long)cardFieldFlag;
          paramPtr->inArgs[1] = fieldID;
          paramPtr->request = xreqGetFieldByID;
    paramPtr->entryPoint();
          return (Handle)paramPtr->outArgs[0];
}


pascal void SetFieldByName(paramPtr,cardFieldFlag,fieldName,fieldVal)
          XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
          StringPtr     fieldName;    Handle        fieldVal;
/* Set the value of field fieldName to be the zero-terminated string
   in fieldVal.  The contents of the Handle are copied, so you must
   still dispose it afterwards. */
{
          paramPtr->inArgs[0] = (long)cardFieldFlag;
          paramPtr->inArgs[1] = (long)fieldName;
          paramPtr->inArgs[2] = (long)fieldVal;
          paramPtr->request = xreqSetFieldByName;
    paramPtr->entryPoint();
}


pascal void SetFieldByNum(paramPtr,cardFieldFlag,fieldNum,fieldVal)
          XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
          short         fieldNum;                                 Handle        fieldVal;
/* Set the value of field fieldNum to be the zero-terminated string
   in fieldVal.  The contents of the Handle are copied, so you must
   still dispose it afterwards. */
{
```

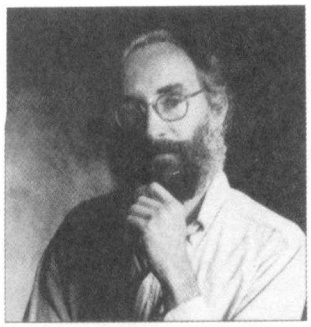
(continued on next page)

Listing Four *(Listing continued, text begins on page 56)*

```
        paramPtr->inArgs[0] = (long)cardFieldFlag;
        paramPtr->inArgs[1] = fieldNum;
        paramPtr->inArgs[2] = (long)fieldVal;
        paramPtr->request = xreqSetFieldByNum;
    paramPtr->entryPoint();
}


pascal void SetFieldByID(paramPtr,cardFieldFlag,fieldID,fieldVal)
        XCmdBlockPtr  paramPtr;     Boolean       cardFieldFlag;
        short         fieldID;                                  Handle        fieldVal;
/* Set the value of the field whose ID is fieldID to be the zero-
   terminated string in fieldVal.  The contents of the Handle are
   copied, so you must still dispose it afterwards. */
{
        paramPtr->inArgs[0] = (long)cardFieldFlag;
        paramPtr->inArgs[1] = fieldID;
        paramPtr->inArgs[2] = (long)fieldVal;
        paramPtr->request = xreqSetFieldByID;
    paramPtr->entryPoint();
}


pascal Boolean StringEqual(paramPtr,str1,str2)
        XCmdBlockPtr  paramPtr;     Str31 *       str1;         Str31 *       str2;
/* Return true if the two strings have the same characters.
   Case insensitive compare of the strings. */
{
        paramPtr->inArgs[0] = (long)str1;
        paramPtr->inArgs[1] = (long)str2;
        paramPtr->request = xreqStringEqual;
    paramPtr->entryPoint();
        return (Boolean)paramPtr->outArgs[0];
}


pascal void ReturnToPas(paramPtr,zeroStr,pasStr)
        XCmdBlockPtr  paramPtr;     Ptr           zeroStr;      StringPtr     pasStr;
/* zeroStr points into a zero-terminated string.  Collect the
   characters from there to the next carriage Return and return
   them in the Pascal string pasStr.  If a Return is not found,
   collect chars until the end of the string. */
{
        paramPtr->inArgs[0] = (long)zeroStr;
        paramPtr->inArgs[1] = (long)pasStr;
        paramPtr->request = xreqReturnToPas;
    paramPtr->entryPoint();
}


pascal void ScanToReturn(paramPtr,scanHndl)
        XCmdBlockPtr  paramPtr;     Ptr *         scanHndl;
/* Move the pointer scanPtr along a zero-terminated
   string until it points at a Return character
   or a zero byte.  */
{
        paramPtr->inArgs[0] = (long)scanHndl;
        paramPtr->request = xreqScanToReturn;
    paramPtr->entryPoint();
}


pascal void ScanToZero(paramPtr,scanHndl)
        XCmdBlockPtr  paramPtr;     Ptr *         scanHndl;
/* Move the pointer scanPtr along a zero-terminated
   string until it points at a zero byte.  */
{
        paramPtr->inArgs[0] = (long)scanHndl;
        paramPtr->request = xreqScanToZero;
    paramPtr->entryPoint();
}
```

End Listings

THE DEATH OF THE MACINTOSH



Michael Swaine is the former editor-in-chief of DDJ *and is presently coeditor of* The Macintosh II Report, *a journal for Mac II users.*

I'm going to miss the little toaster.

The Apple Macintosh that I bought in 1984 now sits on my mother's desk in a small town in the Midwest. Apple has disinherited that Macintosh, killed off the model. Mom tells me it hasn't affected the performance of her machine.

The spot on my desk that was once home to the Macintosh—in fact, a spot considerably larger than was home to the Macintosh—is now occupied by a 35-pound RGB monitor; a keyboard that stays on the desktop because it is too wide to fit on my lap between the arms of my chair; and a large gray box containing a 40-Mbyte hard-disk drive, six slots, and a loud fan.

They call it a Mac, too.

But it's not. A Macintosh is a cute little box of such human dimensions that Berke Breathed could turn it into a character in Bloom County. A Macintosh is an appliance. You can't put a different interface on it, and you can't add internal hardware to it. Can you add cards to a toaster? The only slots in a Macintosh are where you put the bread—I mean, disks. And a Macintosh does *not* have a fan.

I've also had my hands on a Unix-equipped version of this same machine. It is now possible, as detailed in the pages of this very magazine, to write Unix software that calls Mac ROM routines and/or displays a Mac-like user interface. It is possible to launch a Macintosh application on this machine under Unix, and it is possible to develop software that thinks it's Unix software part of the time and Mac software part of the time, possibly displaying the same Mac-like user interface in either case but possibly not. A Macintosh does *not* run Unix.

And it's not just this particular model. Apple has redefined the user interface, adding pop-up and hierarchical menus. Throw in color (which is not restricted to the one model) and the choices in the design of a user interface increase by at least three orders of magnitude. But developers of Macintosh software aren't supposed to have *any* choice in the design of the user interface.

Then there's HyperCard. Bill Atkinson calls it a software erector set, but it's really a virtual machine. It violates the Mac user interface guidelines so fundamentally that it doesn't make any sense to call it a Macintosh program. With HyperCard, Apple is telling all those middle-ground users who don't want to develop application software but who would like to write the equivalent of a batch file that they can't do that on a Mac but they can do it in HyperCard. A Macintosh does *not* support casual programming.

What it comes down to is this: In the past year and a half, Apple has increased the power, flexibility, and potential of the line of machines it persists in calling Macintosh. In doing so, it has attracted software developers in droves, created a new category of freeware/shareware, established a serious beachhead in corporations, impressed the community of business and industry reporters and analysts, and made its stockholders very happy.

I won't begrudge it all that, I guess. But I wish it'd stop calling the machines Macintoshes. They're not Macintoshes.

They're computers.

Michael Swaine

Dr. Dobb's Software Engineering Sourcebook

Winter, 1988

DR. DOBB'S JOURNAL SPECIAL EDITION

COVERING PCs, Macs & Workstations

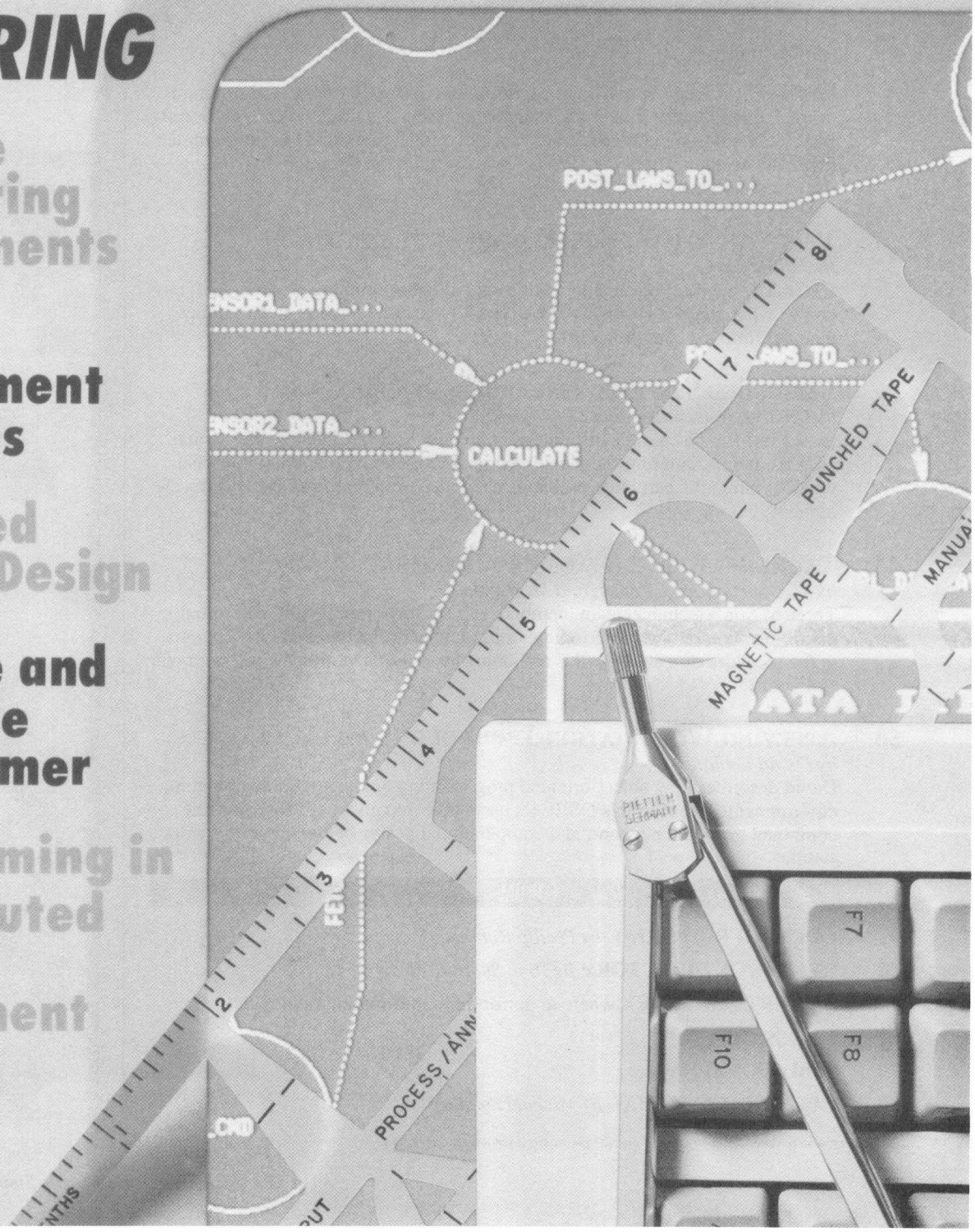
Display Until March 1, 1989 3.95 (4.95 CANADA)

DR. DOBB'S

SOFTWARE ENGINEERING SOURCEBOOK

FEATURING

- Software Engineering Environments
- API Development Platforms
- Embedded Systems Design
- Software and the Single Programmer
- Programming in a Distributed Process Environment



0 38351 17742 3 84

CONTENTS

ARTICLES

8 SOFTWARE ENGINEERING ENVIRONMENTS
by Stowe Boyd
Stowe provides readers with a map of the meandering journey toward a coherent software engineering environment. He also presents the strengths and weaknesses of various languages and environments with respect to their suitability for large programming projects.

18 SOFTWARE AND THE SINGLE PROGRAMMER
by T.G. Lewis
Professor Lewis of Oregon State begins with a discussion of several programming methodologies. This leads to an examination of OSU (the Oregon Speedcode Universe), an intelligent graphic prototyping system that lets programmers construct an application WYSIWYG-style by showing — rather than telling — the computer what to do.

28 USING AN API AS A DEVELOPMENT PLATFORM
by Jeff Parker
Jeff makes the case that an application specific programming environment may be just what you need so that you can spend your time on the content, rather than the form, of your program.

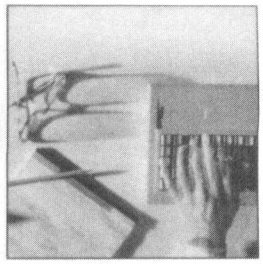
36 EMBEDDED SYSTEMS DESIGN — A SPECIAL CASE?
by David Kalinsky and James Ready
David and Jim discuss the difficulties of splicing CASE tools together which unfolds into a charter for the ideal integrated programming environment that would address the particular problems encountered in real-time systems development.

44 TACKLING LARGE-SCALE PROGRAMMING PROJECTS
by W. Courington, J. Feiber, and M. Honda
Three of Sun's technical team members explain how intelligently controlled distributed computing can bring the unused power of networked workstations to the local desktop and how this technology is only "a motion away" from you and your PC.

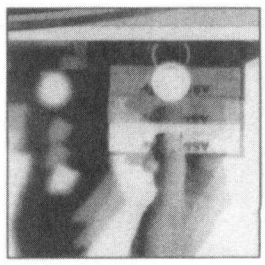
50 APPLYING WORKSTATION TECHNOLOGY TO CASE
by David Leblang
David describes how object-oriented programming and distributed computing environments like Apollo's DSEE let users take maximum advantage of the combined graphics, network, and computational power of contemporary workstations.

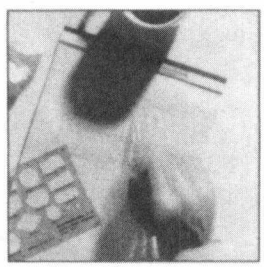
FORUM

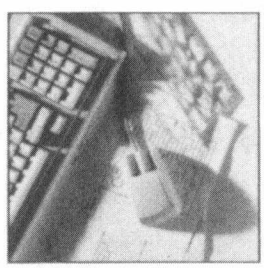
4 OPENING EDITORIAL *by Phillip Robinson*
58 RESOURCE DIRECTORY *by John Scott Ruzicka*
64 ADVERTISER INDEX where to go for more information on products
72 GLOSSARY
76 BIBLIOGRAPHY
80 CLOSING EDITORIAL *by Michael Swaine*

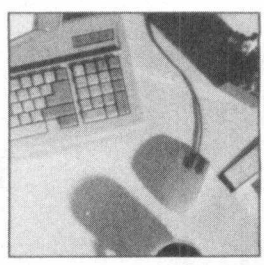

8

18


28


36


44


50

The editors wish to thank Ready Systems Corp. for their assistance with the cover art.

SOFTWARE ENGINEERING

Phillip Robinson is an editor and designer at Virtual Information, a Sausalito, Calif., publishing and programming firm.

The notions of software engineering and CASE (computer-aided software engineering) sometimes get tangled together. They aren't the same thing. CASE is the application of computers to the creation, testing, and modification of software. Software engineering goes beyond that, encompassing not only CASE but the entire process by which a program is imagined, structured, produced, verified, and maintained. CASE tools can certainly organize and speed most any software engineering effort, but they will be most effective if they are themselves designed and used in harmony with proven software engineering techniques.

After all, what is a CASE tool? The term is usually given only to specialized prototyping programs, version-control utilities, and the like. But a debugger is a CASE tool; so is an editor, for that matter, and even an operating system, with its attendant utilities. These are all certainly helpful tools, and the proliferation of sophisticated CASE utilities on the PC level is nothing short of a godsend for personal computer programmers. But the present form of these tools can sometimes obscure the more far-reaching questions of software engineering, such as: How should you organize the many steps and tools of a programming effort? What interface metaphor is most productive? How closely should a particular programming tool match its intended application area? How can large projects be coherently arranged and scheduled to use many computers and many programmers?

These are certainly not idle questions. We devote a staggering number of hours to programming computers and are often left shaking our heads at the inadequate performance that is the result of our labor. Heaps of features and entire applications are left waiting for that imaginary day when we will have the time to really get it right or even get to it at all. Meanwhile, hardware designers from the individual chip to the mainframe system level can boast every year that their products are twice as fast, twice as effective, many times the power at less price than the hardware of yesterday. Software designers are left in the dust, hoping for a few percentage point improvements in programming efficiency. The number of lines of correct code per day expected from a top-ranked programmer is still in the double digits. Combine that excruciating fact with the common wisdom that larger programming teams can have serious communication problems, and you have a recipe for anguish. (You've probably heard the joke, How long will it take twice as many programmers to finish the job? Answer: Twice as long if you're lucky and if they finish at all.)

Nor is the quest for programmer productivity one that only dreamers pursue. The company or individual that comes up with a program in less time can often win the race outright, or at least claim a powerful position in the market. If that program is also easily maintained and enhanced, the company or individual will have a head start every day of the competition. If some of the program's code can be reused in the product that succeeds it, the race will be over almost before it starts. Software productivity may even have an impact at the political level, from economic competition to military conflicts. The U.S. military spends a great deal of money on software. It even demanded a new programming language, armed to the teeth with software engineering ideas and structures, just so it could get its programming house in order. So if it's so important, what about some answers? How do you write good code quickly? A good compiler with a library of reusable modules is vital, as is a flexible editor, a potent debugger, and other such tools. These have long been advertised, reviewed, and otherwise touted in *Doctor Dobb's* and other journals. Other important

tools, however, don't get as much press. Add a prototyper to the stew if you want to get some ideas right away about the program's general design and look. If it's a big project, you might consider adding a version control utility, a documentation librarian, and other such CASE utilities. There are even programs that help you analyze programs for their adherence to standards, and other programs that can draw a flowchart from someone else's program — a real boon if you're stuck maintaining poorly documented code. For some niches there are fourth-generation languages (4GL): programs that can automatically generate application programs from descriptions of the desired input and output.

These utilities are part of one important avenue to improved programming productivity, and this special issue Sourcebook contains a guide to them, listing prices, platforms, and so on. Instead of rooting through the ads at the back of each programming magazine in your library, you can find the companies devoted to this market all collected in one place. Along with that resource guide, we've included a glossary that explains the terms of software engineering and CASE to help you pin down some of the ambiguous phrases and marketese you may encounter when discussing CASE tools. But the resource guide is not enough, by itself, to show where software engineering is going. Nor is fast coding enough to make a good programmer. You should know about research that points the way to huge improvements in productivity — improvements that rocket far ahead of the mere doubling or tripling you might gain from an improved compiler or prototyper. You need to know the limiting implications of today's accepted programming technology and menageries of discrete tools. That's where the articles of this special issue come in. They represent some of the most important directions of software engineering, discussing issues such as programming environments, intelligent prototyping, distributed computing, version control on large software projects, graphics and object-oriented languages, tool integration, and application-specific virtual machines. An accompanying bibliography provides a map to other articles on the same subject.

This issue touches on software engineering of both today and tomorrow, from the embedded real-time system through the personal computer to the network of workstations. We planned this issue so that it will remain useful for years, long past the time when any particular company, program, or review of that program may have become history. We've made a "keeper," a volume that could easily hold a place on your shelves well into the 1990s.

- **Software Engineering Environments** *by Stowe Boyd.* Software engineering and CASE haven't just appeared out of the blue. As computer tools for programming were developed over the years, there have been repeated attempts to assemble those tools into coherent environments. Stowe Boyd discusses the history of these environments and puts today's important software engineering languages and environments — such as Lisp, Smalltalk, Unix, DSEE, and NSE, and even hypertext — into context and explains some of their strengths and weaknesses. His ample bibliography contains pointers to the prominent sources of software engineering research.

- **Software and the Single Programmer** *by T.G. Lewis.* Today's prototyping tools for personal computers — such as Bricklin's Demo Program and the SmethersBarnes Prototyper — are mostly vacuous. They can be helpful in creating example screens

and menus for a program, and for linking those elements together in a simulation of what that program would supposedly do. They don't, however, lead to actual programs or program code. Professor Lewis of Oregon State has been working on Oregon Speedcode Universe (OSU), an intelligent graphic prototyping system which can directly generate a program. He suggests that only this sort of radical change of programming style, away from today's linguistic paradigm, can yield productivity improvements of ten to a hundred times.

- **Embedded Systems Design — A Special CASE?** *by David Kalinsky and James Ready*. Using real-time software development as an example, Kalinsky and Ready discuss the difficulties of putting CASE tools together. They discuss the critical steps of software development — from systems engineering and design through software design, coding, debugging, integration, testing, installation, and maintenance — and how the CASE tools for each of those steps can be linked to their neighboring tools.

- **Using an API as a Developer Platform** *by Jeff Parker*. Parker describes how the proper selection of a programming metaphor, appropriate for a specific application area, or even the construction of a virtual machine on which to program, can have a great impact on programming results. His own firm's LabVIEW is just such a virtual machine for computer-aided test and measurement. Non-professional programmers can put together reusable graphical instrument components that resemble their physical counterparts to quickly build data acquisition and analysis programs.

- **Applying Workstation Technology to CASE** *by David Leblang* and **Tackling Large-Scale Programming Projects** *by William Courington, Jonathan Feiber, and Masahiro Honda*. This pair of articles — from two of the leading workstation makers, Sun and Apollo —talks about what happens to CASE and software engineering when you move beyond a single computer and a single programmer. Large projects with dozens of programmers, hundreds of thousands of lines of code, and thousands of files disintegrate into chaos without robust version control and configuration management software. Today, most such projects are handled on a local area network. That can complicate the control task. For example, the system must deal with parallel development — concurrent access to a single file by programmers who want to make different changes. It can also aid the programmers. For instance, the unused power of other workstations can be brought to bear on a local problem through intelligently controlled distributed computing. These articles discuss the issues and the competing products — Sun's NSE and Apollo's DSEE. This technology is already reaching into the realm of personal computers and will be critical in any large development effort of the 1990s.

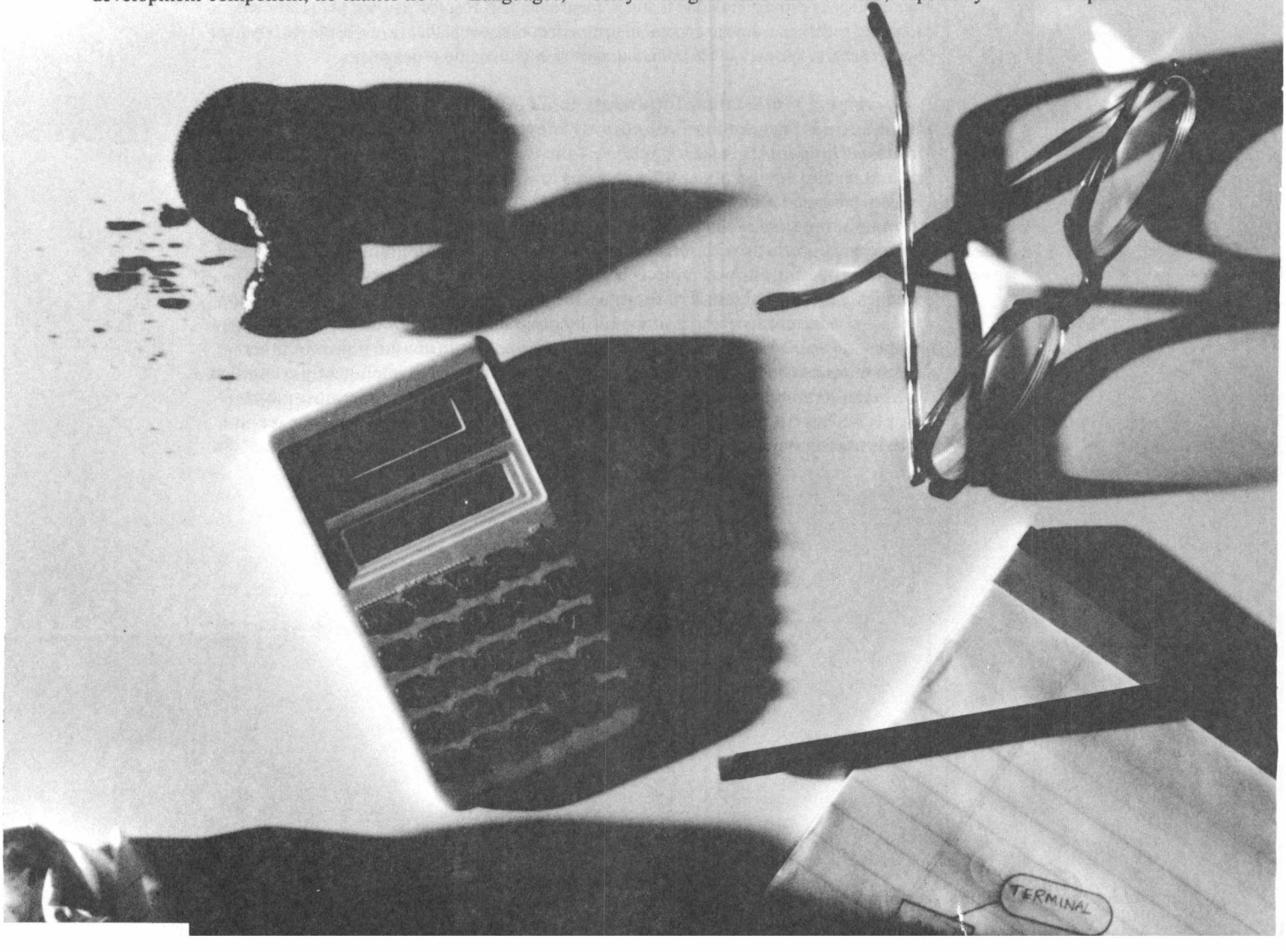
Software Engineering Environments

What has inspired the CASE phenomenon?
What comes after CASE and Software tools?

by Stowe Boyd

Some programmers argue that contemporary programming languages are simply inadequate for developing large software systems. They maintain that current implementations of programming languages may actually hinder more than help in the construction of reliable, verifiable, or even manageable software. Their point is that any software development component, no matter how good it is by itself, is compromised if it doesn't work well in combination with other tools. This perception has led to the development of integrating/integrated software, such as code documenters, analyzers, librarians, interactive debuggers, and more, designed to address the problems of large scale software development.

In his paper "Beyond Programming Languages,"[1] Terry Winograd states that "the main activity of programming is not the origination of new independent programs, but the integration, modification, and explanation of existing ones." Winograd goes on to point out that the intellectual activity associated with software development is largely one of gaining insight. Only a small proportion of time is dedicated to generating new software, especially when compared to all



other activities. This is not a failure of environments or programming languages but rather an inescapable reality.

At some ill-defined point, a set of programming tools that offers a sufficiently high level of integration may become collectively known as a software engineering environment. This article examines a handful of development environments that were selected on the basis of their historical impact, their degree of use, or their promise for the future. These criteria are examined in light of three key themes: point of view, structure, and scope.

Point of View

A software development environment exists to increase the programmer's awareness of the program under development. This understanding may be limited to the static and dynamic aspects of the software backplane or may extend beyond that, representing a model of the software development process itself, including development phases, organizational structure, and the interrelationships between projects.[2]

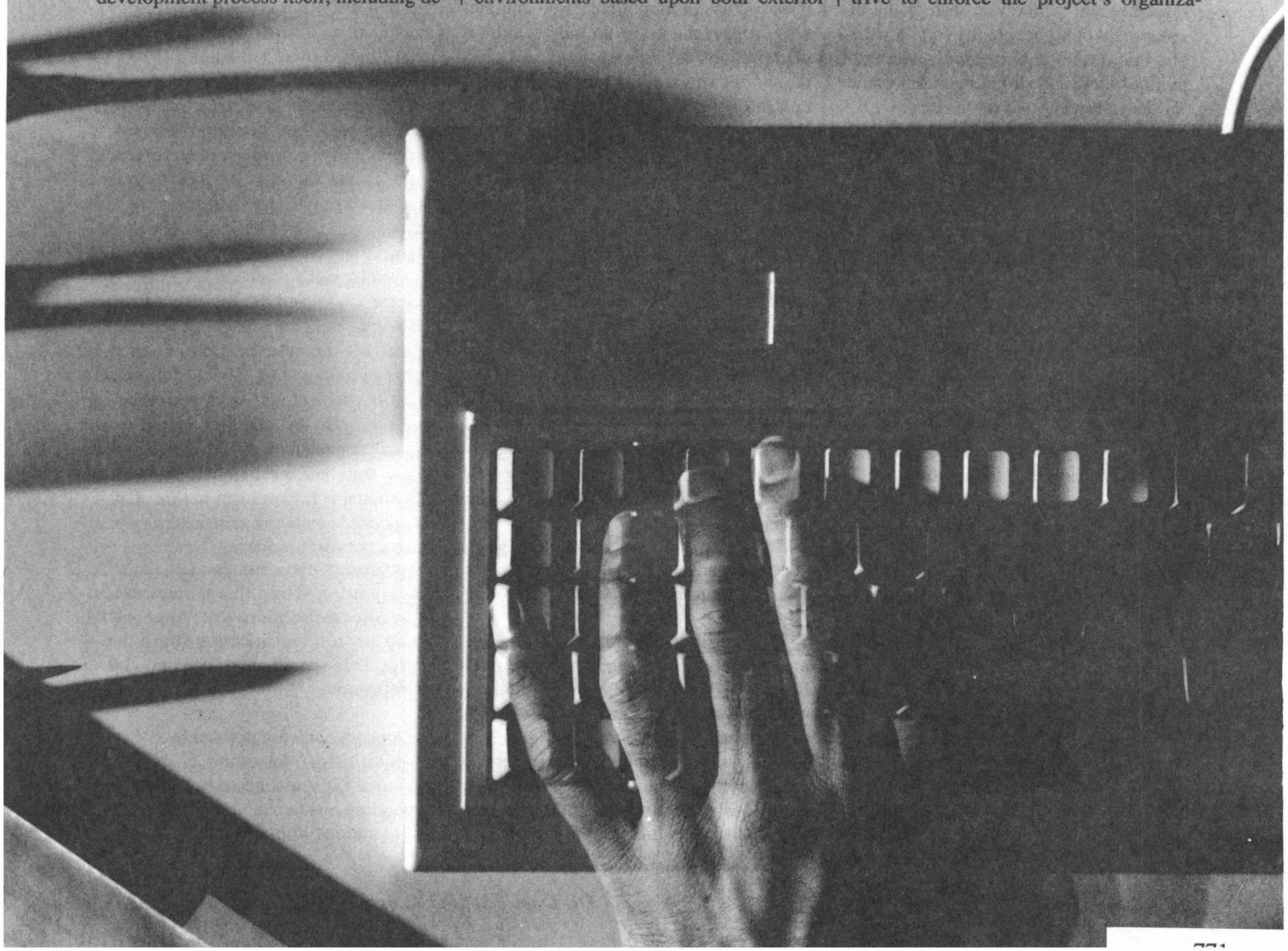
Ultimately, the user's point of view dominates all evaluation of environments. The bias expressed in this article is a corollary to Winograd's observation that "An environment serves to the degree that it assists in the reuse of software first by providing insight into the software and its workings and structure, and then by providing means to modify and manage software systematically and the environment's success is directly proportional to the naturalness, uniformity, and generality of the user's point of view."

Users' concerns focus on exterior aspects of a system, such as the user interface, performance, capabilities, and user modifiability. But there are other perspectives. Tool builders, for example, have broader concerns, which revolve around system structure. Management evaluates environments based upon both exterior aspects and system structure because these characteristics influence flexibility and productivity, but they are more concerned with the scope of the environment.

Structure

The natural complement of point of view is environment structure, which reflects the system's architecture. An environment's point of view sometimes provides a limited reflection of the system's structure. In general, however, this is not the case — in fact much of what goes on "below the hood" is never seen by the user.

Environment components form a framework that supports software development. The effectiveness of a framework cannot be judged abstractly. First, it must contain the required tools. It must also be robust enough to support software while it is undergoing development. Finally, it must focus user activity in ways that contrive to enforce the project's organiza-



tional requirements. This dimension — the environment's dynamic nature — is environment scope.

Scope

Ask any three software engineers if software has a life cycle, and they would probably agree that it does. Ask them to describe the life cycle, and you will probably receive three different answers. Their answers would depend upon many factors: organizational policies, training, experience, and psychology. It is unlikely that you will receive one expressive, generalized definition of the software life cycle.

Still, software is both the result and a record of the complex development process. This process comprises, but is not limited to, activities such as design, coding, testing, and documentation. The span of activities that are supported by an environment thus defines its scope. For example, an environment that provides capabilities for automated test generation and measurement is likely to lead to the development of higher quality software than one that does not. Scope, however, trails point of view and structure in importance for the following two reasons:

1. A well-designed environment with a powerful point of view is likely to be both adaptable and extendable in scope.
2. Broad scope may be unused if it is not well integrated or if it does not mesh with the user's point of view.

Environment Research and Development

Describing environments can be something like dissecting small animals: too many cuts and there is nothing left to examine. For this reason the following discussion addresses the major trends in environment research over the past few decades, making only a few cuts into the specific environments mentioned and focusing on major contributions.

Stowe Boyd is director of Research and Development at Meridian Software Systems where he is currently developing the Meridian Software Development Platform, a distributed environment based on HyperText principles. He has published numerous articles on the subject of software development environments and related topics.

Early research and development in environments went on for some time prior to the adoption of the term *environment*. These early efforts were generally concerned with a specific problem such as the development of system software in the Unix system,[3,4] experimental programming in Cedar/MESA,[5,6,7] or exploratory programming of Lisp programs in InterLisp.[8]

It has often been noted that everyone has a programming environment; generally it is ad hoc and not coherent. If an operating system with a bunch of tools is an environment, why go any further? One example of an operating system that has gradually evolved into an environment is the Unix system and its many descendants and variants.

The Unix System As an Environment

The Unix operating system[3] was invented in 1969 by Ken Thompson, and by 1979 it was in use in more than 2,300 computers.[4] Today, there are literally hundreds of thousands of Unix systems in use in the world, and it has become not only a de facto operating system standard but also a proposed standard.[9]

The Unix environment was originally devised to support tool building, and therefore its point of view is that of a tool builder, not a general software developer. Its basic structure is largely visible to the user and has been characterized as follows:[4]

- A hierarchical file system incorporating demountable volumes
- Compatible file, device, and interprocess I/O
- The ability to initiate asynchronous processes
- System command language selectable on a per-user basis
- More than 100 subsystems, including a dozen languages
- A high degree of portability

The contributions of the Unix environment are significant. To some extent this is because of the nature of the C programming language and its broad use in the environment. Unix serves Winograd's argument well by supporting the reuse of C software. Direct reuse is provided by Unix C libraries, which in turn provide access to system functions and utilities.

More abstract reuse occurs in the form of code generators, such as YACC and Lex.[10,11] Software designers can apply these source code generation tools to the tasks of parsing textual input simply by specifying the input's form. This form of reuse support has driven work in diverse areas such as compilers and user interface systems[12,13] and has structured language-specific environments within Unix.[14] The Gandalf project at Carnegie-Mellon[15] and the synthesizer generator project at Cornell[16] have built upon these ideas. This has led to syntax-oriented environments in which editors, compilers, and other tools share a common representation of software.

These ideas continue to be applied today in projects such as the Distributed Ada Programming Support Environment effort[17,18] and the Anna project,[19] which are dedicated to distributed development and the formal specification of Ada systems, respectively.

The Unix filter paradigm of program construction — small, single-purpose programs linked by I/O channels — has had a major influence on tool building. The ODIN/Toolpack[20] and Arcadia[21] projects have adopted the filter approach and extended it by considering environment tools as assemblages of tool fragments that are brought together for a specific purpose and linked by interprocess communication channels.

The original Source Code Control System (SCCS)[22] and the related Make program[23] defined an extremely serviceable model for the specification of software system composition. The SCCS system is a Unix toolset that manages versioned text files, provides the user with a means to incorporate changes into software systems, and manages related documents. The model is hierarchical in that changes can be made away from an arbitrary baseline; a version-numbering scheme is provided based upon the fundamental tree form. *make* is a tool that derives executable software systems (or components) based upon a *specification* (the "makefile"). This specification has two classes of definition:

1. Dependencies are defined by file suffix relationships — for example, stack.c is a C source file and stack.o is an object file depending on stack.c.
2. Operations implement the derivations

associated with dependencies — for instance, *cc -o stack.o stack.c* represents the derivation of the stack.o object file from the stack.c source by invocation of the C compiler.

In response to the problems inherent in the SCCS/Make approach, an enormous amount of work has gone on in the area of software composition. Revision Control System (RCS)[24] is a counter to the incompatibilities in SCCS/Make that incorporates several more efficient and powerful features, such as backward version maintenance, retrieval of file versions by multiple keys, and deferral of specific versions.

Apollo's Domain Software Engineering Environment (DSEE)[25] represents a major departure from the basic SCCS/Make approach. DSEE supports programming in the large across a distributed network of Apollo computers. Software composition in DSEE distinguishes the hierarchic system model from the configuration. Composition is source-limited; all objects are managed transparently by DSEE in an "object pool" that is shared *among* users. Versions and variants of software are maintained separately. Versions are more or less equivalent to SCCS or RCS versions, but DSEE variants also have novel features, including alternate implementation of system elements (different floating-point libraries for different machines, for example) and conditional compilation.

The actual construction of a DSEE system binds a system model to a specific version set using a "configuration thread" (CT). A CT comprises a series of rules for determining the actual set of versions to be used; evaluation of the rules results in a bound configuration thread. It is this bound form that derives the actual executable software. Shared objects appropriate for inclusion in the executable are found prior to duplication, minimizing construction effort.

DSEE shifts the Unix point of view from a tool builder's to a true software developer's. Other more recent efforts, such as the Sun Network Software Environment (NSE),[26] represent the awareness of the need to support software composition in a uniform, reliable, and robust way without forcing the developer to be a tool builder. This system automatically updates information used to manage system composition and, like DSEE, closely integrates version and variant control with system build activities.

The second and potentially greatest effect of the widespread adoption of these systems is a shift in the Unix structure. NSE extends the Unix information base to include versions and several other structural changes of significant scope. These represent a point of departure from the traditional Unix paradigms of TTY-type terminals, simple time sharing, and the filter approach to information sharing across tools. Unix has made a large contribution, but it is not enough.

Unix is an environment geared to the composition of small- to medium-sized systems and the construction of tools. Efforts such as DSEE and NSE represent a new phase for Unix as a software engineering environment for large systems. It is noteworthy that Unix software composition capabilities are sufficient to manage the complexities of Unix itself — all 500,000 lines of it — and have proven useful for the management of medium-sized business software systems as well.[27] When confronted with systems comprising millions of source-code lines, however, other approaches are required.

The Ada Language System,[28] the WIS Software Development and Maintenance Environment,[29] and the Rational Environment[30] are attempts to support large-scale development for Ada systems. Ada is designed for the construction of large, reliable, maintainable, embedded systems and needs the support of an appropriately devised environment.[31] Note that this is not the fundamental character to the language but an aspect of the character of large systems.

Experimental and Conceptual Programming

Lisp is one of the earliest programming languages and one of the most hardy.[32] Sandewall[33] characterizes the use of the language quite well: "Lisp is used almost entirely as a research tool. The average Lisp user writes a program as a programming experiment, that is, in order to develop the understanding of some task, rather than in expectation of production use of the program. The act of developing the program, not the act of running it (even for test data) constitutes the experiment. As a consequence, the program is likely to be large and complex, to undergo drastic revisions while it is being developed, and to be thrown away before it has been 'completed' by conventional programming standards since it will already have served its purpose before then."

Very early it became evident that experimental programming constituted a radi-

Environmental Structure

Environmental structure can be characterized by answers to the following questions about its components. For the purposes of this article hardware is generally excluded.

- Information base — What mechanisms exist to manage environment information? Common approaches include hierarchical file systems, relational databases, and persistent object managers. Are the fundamental units of environment information, or objects, typed?
- Specification — How does the environment manage the attributes of software, such as system composition, intercomponent relationships or dependencies, and access constraints? Are different specifications used, or is a uniform approach used?
- Context — What methods exist to collect system components into information domains or contexts? Are projects hierarchically decomposed, or are graph structures used? Are tools and users gaining access through equivalent context systems?
- Actions — How do information changes take place? Are all changes associated with explicit execution of tools, or are some changes implicit, triggered by constraints?
- Distribution — What mechanisms support the distribution of information and processing over distributed hardware?
- Integration — How are tools integrated? Must alien tools be rewritten to work in the environment? Do tools share common information? Can tools act in concert, and if so, how are these toolsets composed?
- User interface — Is there a structural separation between the display device's characteristics, operational access to information, and the internals of tools? — *S.B.*

cally different point of view than was then current. Indeed, in 1969 Teitelman[34] introduced the concept of a programming environment: "The programmer's environment influences, and to a large extent determines, what sort of problems he can (and will want to) tackle, how far he can go, and how fast he'll get there. If the environment is cooperative and helpful (the anthropomorphism is deliberate), the programmer will be more ambitious and productive. If not, he will spend most of his time and energy fighting a system that at times seems bent on frustrating his best efforts."

Lisp users were atypical at that juncture: they were committed to increasing programmer productivity by the application of computer resources and believed in the development of sophisticated tools rather than simple, multipurpose ones. This was both because of the point of view of the developer as experimenter and the intense scope of their programming environment.

The single most influential experimental environment effort of the 1970s was InterLisp.[34] Founded at Bolt Baranek and Newman, and later pursued at Xerox PARC, InterLisp represents the expert-oriented environment. The environment is geared to exploratory programming in Lisp, but the most important contributions of the work are not Lisp specific:

- DWIM — The Do What I Mean (DWIM) facility can be incorrectly viewed as the system facility that corrects users' errors and attempts to guess users' intents. In fact, it is a pervasive philosophy about user interface design.
- Masterscope — This is an interactive program for analyzing and cross-referencing users' programs. Masterscope is specifically geared to helping the users predict the impact of changes. The program maintains a database of analyses, and users can formulate queries to gain information about system dependencies.
- Programmer's Assistant — The principle behind the Programmer's Assistant is user interaction with an active intermediary instead of a passive editor. With now-common features such as undo and a history log, the system becomes the progenitor of contemporary programming systems. The fix feature, which allows users to amend the history log and reexecute the commands, is a powerful step forward. Later work on programmer's apprentices[35,36,37] has its roots in InterLisp.

The treatment of programs as data structures manipulated within a large address space is key to many later approaches to environments. All work on grammar-based environments, particularly the grammar-driven and grammar-generated environments (such as the Cornell Program Synthesizer[16] and the Gandalf ALOE system),[15] extend the start made in InterLisp.

Smalltalk

Another branch of research at Xerox has led to a worldwide revolution in computing. Smalltalk, a programming system that has evolved over a decade into Smalltalk-80,[38,39] is based upon the principles of object-oriented programming: objects and messages.

As defined by Goldberg,[39] "An object is a uniform representation of information that is an abstraction of the capabilities of the computer to store information. An object has the capacity to store information; we say that an object has a private memory. An object also has the ability to manipulate its stored information or to carry out some activity. These are called the operations of the object. The set of operations is referred to as the object's interface."

Objects communicate by message pass-

Software Spreadsheets

One interesting approach to environment support is based on the notion of computation media. These are more or less "software spreadsheets" in which programming is both textless and naively understandable. One example of this approach is Boxer.[48]

People have a lot of common-sense knowledge about physical space that can be used to make computers more comprehensible. The spatial metaphor encourages people to interpret the organization of the computational system in terms of spatial relationships. Using a Boxer system is like moving around in a large two-dimensional space. All computational objects are represented in terms of boxes, which are regions on the screen that contain text, graphics, or other boxes. Naive realism is an extension of the "what you see is what you have" idea that has become commonplace in the design of text editors and spreadsheets but, unfortunately, not for programming languages. The point is that users should be able to pretend that what they see on the screen is their computational world in its entirety.[48]

Boxer, along with similar systems such as HyperCard, provides a dynamic computation system in which the user alternates between the view of programmer and user, creating, extending, and customizing computational structures. In a real sense it can be viewed as a textless return to InterLisp or Smalltalk.

A more powerful but similar paradigm is that of Hypertext or Hypermedia.[49,50,51,52] Ted Nelson, one of the pioneers of Hypertext, once defined it as "a combination of natural language text with the computer's capacity for interactive branching, or dynamic display — of a nonlinear text — which cannot be printed conveniently on a conventional page."[50]

A Hypertext system structures information by linking individual documents. A software development process may be viewed as the production of an interrelated series of documents. These documents include operational concept documents, reference manuals, test plans, acceptance test results, review notes, source code, graphical designs, and a myriad of other documents.

The Hypertext system provides a dynamic and flexible means of coordinating this information that takes the place of traditional, inflexible database-centered techniques. The author is designing an environment based on an integrated collection of tools (including an Ada compilation system) that supports the development of Ada software systems based on the Hypertext model. The Hypertext links can be exploited by environment tools as well as software engineers. A configuration management tool, for example, will use links created by the Ada compiler's library manager. This system will be more closely coupled to a compilation facility than other projects, such as that mentioned by Bigelow.[51] — S.B.

ing: "An object carries out one of its operations when another object sends it a message to do so. Each object knows the messages it can understand; associated with it is a procedure or method that describes how the object should answer the message."[39]

However important object-oriented programming may be or may come to be, the Smalltalk point of view has been much more influential. Smalltalk provides multiple views of the software under development. These include browser views (used to read object descriptions, to modify operations, and to perform change and version management) and inspector views (for discovering the state of objects during debugging).

Smalltalk's scope is tightly coupled to the exploratory development of Smalltalk-80 programs. Reuse of existing software is supported in a complex and powerful manner. All Smalltalk-80 objects are instances of an object class. A class is an abstraction that allows the description of common characteristics. Smalltalk-80 programming consists of creating classes and their instances and specifying sequences of messages to these instances.

New classes are generally derived from existing classes by inheritance of the parent classes' characteristics with new extensions. The purpose of much of Smalltalk-80 is supporting users in understanding existing class definitions (built-in explanations and example messages, for example) and extending them in novel ways.

Coupled with high-resolution workstation technology, Smalltalk's point of view has evolved into the standard spatial metaphor, as popularized by the Macintosh. Small but informative touches such as the changing of the cursor shape to denote current activity are used in Smalltalk-80 to provide consistent feedback to the user. The development of "nonintrusive" editors allows a nonmodal form of interaction. The user points to an area in which something should happen, selects that area, and then chooses an operation from a menu. Simply by pointing to another area, the user can signal a change about what actions to take.[39]

In retrospect, Smalltalk-80 is the source of the contemporary lingua franca for user interaction with environments and for computer interaction in general. The concept of multiple views over a collection of software components is perhaps the most basic information structuring concept to come from this work. Indeed, it has led to the development of conceptual programming systems.[40,41]

Although Smalltalk-80 permits the exploratory development of relatively complex systems, the complexities introduced are generally structural — class hierarchies and complex message mechanisms — rather than complexities because of size.

Programming in the Large

Cedar is an environment research project at Xerox designed to support exploratory programming.[5,6,7] The project was not principally oriented toward environment research, and therefore its structure is often conservative in design. Although it was not intended for very large systems, several of its principal structural features in fact mirror the requirements for programming in the large.

Cedar is both an environment and a language; the language is a superset of the Mesa language. Mesa introduced a few language features that are fundamental to the development of reliable large systems, most notably the separation of interface and specification of programming modules. Cedar differs from Mesa primarily by the inclusion of collectible storage, freeing the programmer from responsibility for management of free storage. These attributes are critical because Cedar is basically an open system and all applications operate in a single, large address space.

Cedar represents one of the first environments in which a significant commitment was made to capitalize on each developer having a high-performance, high-resolution workstation. This is a central component of the environment structure. Just as important to the user's perception of the environment is the ability to "multitask" and to switch rapidly among several different tasks. The Cedar system allows for the efficient creation and management of many independent processes, and system applications are designed to be nonintrusive so that users may determine the most effective use of their time.

Several other aspects of the system structure deserve mention:

- DF files — Description files, or DF files, are used in Cedar to describe all the components of a system. They are used to retrieve or back up all components referenced. Cedar checks that the contents of a DF file are in fact consistent and complete.
- MakeDo — The Cedar version of Make, MakeDo performs whatever compilations and binds are necessary to create an up-to-date version of the system under development.
- Tioga — Cedar is a system built with Winograd's observations in mind: Users spend more time reading programs than writing them. So instead of a syntax-oriented editor, Cedar is coupled with a complete document-preparation system called Tioga. A great deal of care is given to program format and presentation, with a standard style for many users. This approach accords well with Knuth's arguments for literate programming[43] and represents a departure from reading printouts of programs.

Although Cedar is based upon fast machines, a modular programming language, and a well-integrated toolset, it is the Tioga philosophy of program presentation and printout-free software development that is its greatest contribution.

Arcturus[44] is an environment project geared to programming in the large. Like Cedar, it is based upon an extremely modular language, Ada; is designed for high-performance, high-resolution workstations; and permits a wide range of user-definable presentation mechanisms to help users understand the purpose of source code. Arcturus accommodates the point of view of the large-system builder and at the same time allows for rapid prototyping of programs, based upon the ideas of Lisp Programmer's Assistants, as in InterLisp.

Note that Arcturus is founded upon the notion that "paper is the wrong container for documentation." Program documentation is considered as critical in Arcturus as in Cedar: ". . . documentation must play different roles for different audiences. Depending upon the experience and knowledge of the reader, documentation should reveal appropriate facts — what is appropriate to one reader may be either boring, obvious, and condescending to another, or completely beyond the intellectual grasp of another.

"What is needed is a database in which program forms at various levels of refinement have attributes which lead to comments, whereupon various computed

views can in turn make these comments selectively visible."[44]

Programming in the large is the clearest example of where contemporary environment technology fails. Despite the efforts of hundreds of researchers and the expenditure of hundreds of millions of dollars, no solution to the problems encountered with large systems has been discovered or invented.

Now and the Future

There is promise in new and novel approaches and in the coupling of these emerging approaches with traditional mechanisms. Just as encouraging is a growing awareness in the boardrooms of America that software requires capitalization, that it is not developed in an assembly-line fashion, and that the primary cost factor of software is maintenance.

Application of widely accepted standards to software engineering environments will have a major influence on software productivity. Not only will standardization of environment services such as Posix and X Windows allow for portability of tools and applications, but it will also permit a rapid growth in the development and use of reusable software components. The goal of a common environment base — such as the Common APSE Interface Set[45] and Portable Common Tool Environment[46] — seems almost in reach, and commercial versions of these environment databases are already available.

The primary barrier to developing reliable software is the difficulty involved in gaining insight into the software so that it can be reused effectively. It is my belief that software engineering environments succeed to the degree that they provide insight by analysis tools, information structures, flexible user interface systems, and documentation systems. The environments of today often work in spite of their designer's goals or work poorly.

Future environments will obviously exploit the exponential growth in computational power made available by hardware advances and will likewise seek to exploit the sudden availability of network connectivity. Perhaps not so obvious is the need to break a "complexity barrier"[47] to help users make effective use of these systems. It is exactly this help that software engineering environments must give.

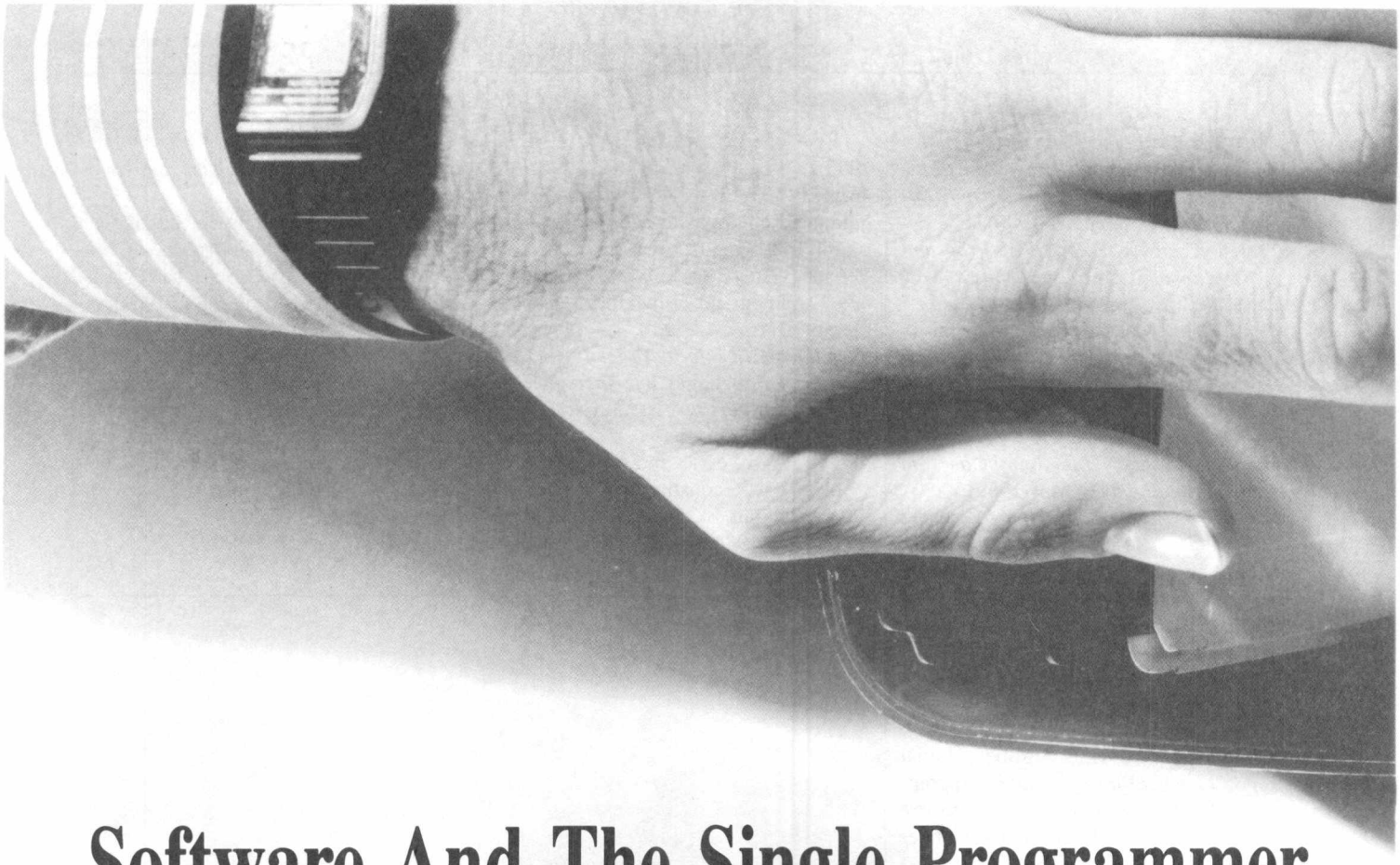
Notes

1. T. Winograd, "Beyond Programming Languages," *CACM*, vol. 22, no. 7 (July 1979).
2. L. Osterweil, "Software Processes Are Software, Too," *Proc. Ninth Int. Conf. on Software Engineering, Monterey, Calif.* (March 1987).
3. D.M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Bell Systems Technical Journal* 6 (1978).
4. R.M. Mitze, "The Unix System As a Software Engineering Environment," *Software Engineering Environments*, edited by H. Hünke (North-Holland, 1981).
5. R.E. Sweet, "The Mesa Programming Environment," *SIGPlan Notices*, vol. 20, no. 7 (July 1985).
6. D.C. Swinehart, P.T. Zellweger, and R.B. Hagmann, "The Structure of Cedar," *SIGPlan Notices*, vol. 20, no. 7 (July 1985).
7. J. Donahue, "Cedar: An Environment for Experimental Programming," *Integrated Project Support Environments*, edited by J. McDermid (London: Peter Peregrinus, 1981).
8. W. Teitelman and L. Masinter, "The InterLisp Programming Environment," *Computer*, vol. 14, no. 4 (April 1981).
9. *POSIX Explored* (Santa Clara, Calif. /usr/group, 1987).
10. S. Johnson, "YACC: Yet Another Compiler Compiler," *Berkeley UNIX Manual* (Berkeley, Calif.: 1978).
11. M.E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator," *Berkeley UNIX Manual* (Berkeley, Calif.: 1978).
12. B. Stroustrup, *The C++ Programming Language* (Reading, Mass.: Addison-Wesley, 1986).
13. C. Chedgey, "Papillon — A Support Environment for Graphical Software Development," *ESPRIT Project (496) Report* (1987).
14. S. Boyd, "SYSTANT: An Integrated Programming Environment for Modular C Under Unix," *Proc. 1984 USENIX Summer Conf.* (June 1984).
15. A.N. Habermann, "The Gandalf Research Project," *Department of Computer Science Research Review 1978-1979, Carnegie-Mellon University* (1980).
16. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *CACM*, vol. 24, no. 9 (September 1981).
17. S. Boyd, "Status of the DAPSE Project: A Distributed Ada Programming Support Environment," *ACM SIGSoft Software Engineering Notes* (May-June 1987).
18. S. Boyd, M. Marcus, and K. Sattley, "Extensibility in an Ada Programming Support Environment," *Proc. Nat. Ada Conf., Washington, D.C.* (March 1988).
19. D. Luckham, et al., "Anna Environment," *Software Engineering Notes* (September 1987).
20. G.M. Clemm and L.J. Osterweil, "The Integration of Toolpack/IST," *IFIP Working Conference on Problem Solving Environments for Scientific Computing, Sophia-Antipolis, France* (June 1985).
21. R.N. Taylor, et al., "Arcadia: A Software Development Environment Research Project," *IEEE Transactions on Software Engineering* (1986).
22. M. Rothkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. 1, no. 14 (December 1975).
23. S.I. Feldman, "Make — A Program for Maintaining Computer Programs," *Software — Practice and Experience*, vol. 9, no. 4 (April 1979).
24. W. Tichy, "Design, Implementation, and Evaluation of a Part 3 Revision Control System," *Proc. 6th Int. Conf. on Software Engineering, ACM-IEEE* (September 1982).
25. D.B. Leblang and R.P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proc. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments* (May 1984).
26. E. Adams, et al., "The Sun Network Software Environment," submitted to *SIGSoft Software Development Environments* (1988).
27. T.A. Dolotta, R.C. Haight, and J.R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Systems Journal*, vol. 57, no. 6 (July – August 1978).
28. R. Thall, "Large-Scale Software Development with the Ada Language System," *SofTech Report* (1980).
29. S. Boyd, "The WIS Software Development and Maintenance Environment," *Proc. of Sixth AFCEA Europe Symposium, Brussels, Belgium* (October 1985).
30. *Rational Environment Product Literature* (Mountain View, Calif.: Rational, 1987).
31. J. N. Buxton and L.E. Druffel, "Ra-

tionale for Stoneman," *Proc. Fourth International Computer Software and Applications Conf.* (October 1980).
32. J. McCarthy, et al., *LISP 1.5 Programmer's Manual* (Cambridge, Mass.: MIT Press, 1962).
33. E. Sandewall, "Programming in an Interactive Environment: The LISP Experience," *ACM Computing Surveys*, vol. 10, no. 1 (March 1978).
34. W. Teitelman, "Toward a Programming Laboratory," *Int. Joint Conf. on Artificial Intelligence* (May 1969).
35. W. Teitelman, "A Display-Oriented Programmer's Assistant," *Interactive Programming Environments*, edited by Barstow, Shrobe, and Sandewall (New York: McGraw-Hill, 1984).
36. R.C. Waters, "The Programmer's Apprentice: Knowledge-Based Program Editing," *IEEE Transactions on Software Engineering*, vol. 8, no. 1 (January 1982).
37. C. Rich and H.E. Shrobe, "Initial Report on a LISP Programmer's Apprentice," *IEEE Transactions on Software Engineering*, vol. 4, no. 6 (November 1978).
38. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, (Reading, Mass.: Addison-Wesley, 1983).
39. A. Goldberg, "The Influence of an Object-Oriented Language on the Programming Environment," *Interactive Programming Environments*, edited by Barstow, Shrobe, and Sandewall (New York: McGraw-Hill, 1984).
40. S.P. Reiss, "An Object-Oriented Framework for Conceptual Programming," *Research Directions in Object-Oriented Programming*, edited by B. Shriver (Cambridge, Mass.: MIT Press, 1987).
41. R.V. Rubin, E.J. Golin, and S.P. Reiss, "ThinkPad: A Graphical System for Programming-by-Demonstration," *IEEE Software*, vol. 2, no. 2 (March 1985).
42. C. Green, et al., "Report on a Knowledge-Based Software Assistant," *Technical Report, Kestrel Institute* (June 1983).
43. D.E. Knuth, "Literate Programming," *Computer Journal* (May 1984).
44. T.A. Standish, "Arcturus: An Advanced Highly Integrated Programming Environment," *Software Engineering Environments*, edited by H. Hünke (North-Holland, 1981.)
45. Military Standard Common APSE Interface Set (CAIS), proposed MIL-STD-CAIS, Department of Defense (January 1985).
46. IEPG-TA13 PCTE Evolution Programme, Requirements and Design Criteria for Tool Support Interface (EURAC), Version 3 (July 1987).
47. T. Winograd, "Breaking the Complexity Barrier (Again)," *Proc. ACM SIGPlan SIGIR Interface Meeting in Programming Languages Informational Retrieval, Gaithersburg, Md.* (November 1973).
48. A.A. diSessa and H. Abelson, "Boxer: A Reconstructible Computational Medium," *CACM*, vol. 29, no. 9 (September 1986).
49. J. Conklin, "Hypertext: An Introduction and Survey," *Computer* (September 1987).
50. T. Nelson, "Getting It Out of Our Systems," *Information Retrieval: A Critical Review*, edited by G. Schechter (Washington, D.C.: Thompson Book, 1967).
51. J.H. Bigelow, "HyperText and CASE," *IEEE Software* (March 1988).
52. S. Boyd, "HyperText As a Paradigm for Software Engineering Environments," *Position Paper, Proc. CASE '87 Conf., Boston, Mass.* (May 1987).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**



Software And The Single Programmer

The evolution of programming from telling to showing

by T.G. Lewis

I have been in computing long enough to remember when Fortran, Algol, and Cobol were claimed to be the greatest breakthroughs known to programmers. Even in those days, there were the believers and the nonbelievers: the programmers who scoffed at Fortran because it was too slow and inefficient and the managers who quickly accepted certain inefficiencies because of reduced time to market, lowered development costs, and improvements in reliability.

Since those glory days, little has changed in the way software is handcrafted by artisan programmers. It seems we have reached a productivity plateau — a leveling-off of programmer productivity that should embarrass any programmer aware of the gains made by other knowledge workers. To make things worse, today's software systems are characterized as large and complex, thus requiring time and effort far beyond the applications developed ten years ago. (Recent PC applications require megabytes of RAM and hard disks to load and execute.) What is the single programmer to do?

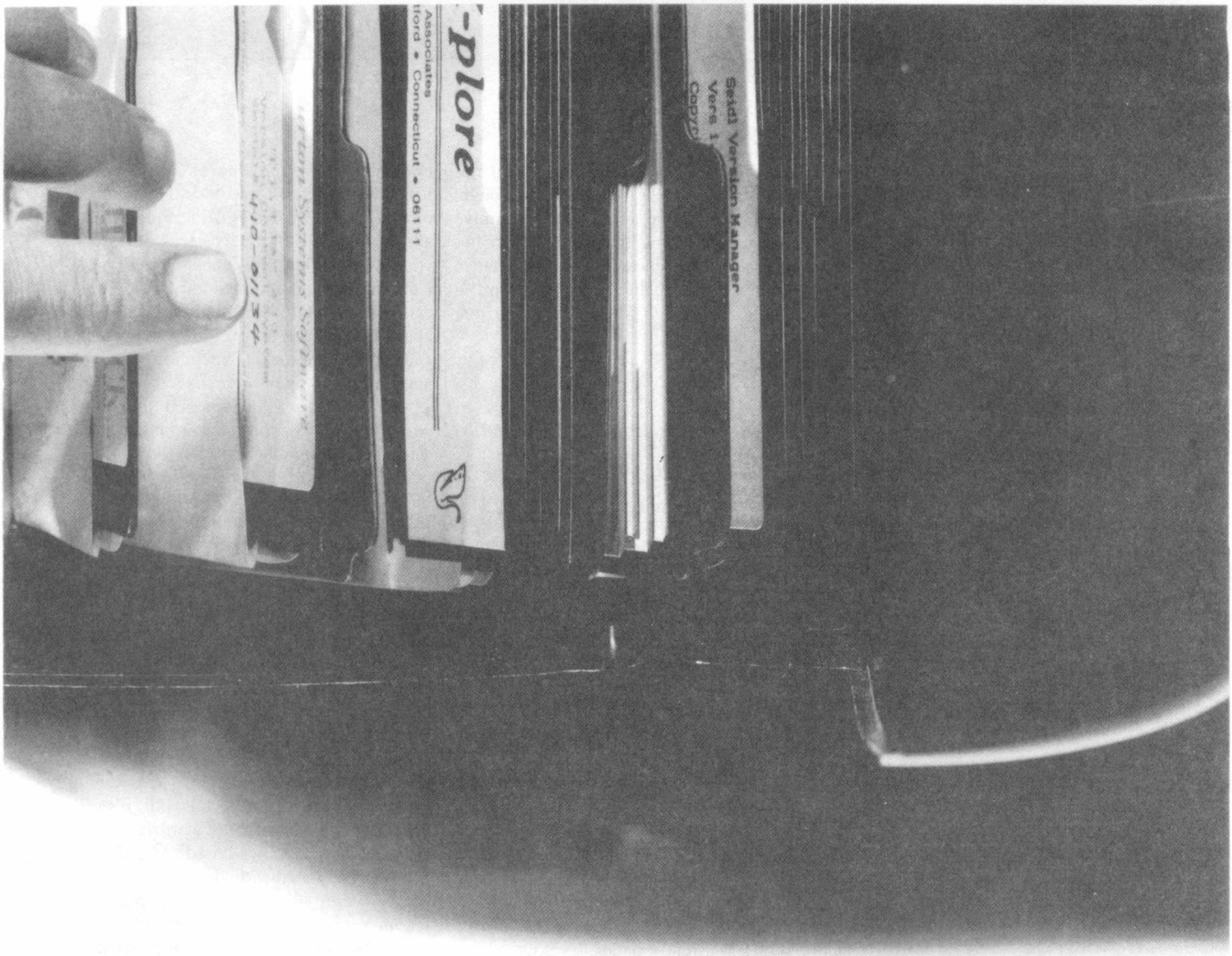
CASE (computer-aided software engineering) tools offer individual programmers a powerful lever for producing large applications quickly. CASE represents a quantum leap in power through software development techniques such as computer-assisted programming, automatic user-interface generation, fourth-generation-language (4GL) application development, and automatic code generation. CASE has desirable side effects: It reduces the drudgery of documentation, improves communication between programmers and customers, and increases software quality. In short, it is a way for programmers to do their job faster and better.

Ted Lewis is professor of computer science at Oregon State University, editor-in-chief of IEEE Software *magazine, and a governor of the Computer Society of the IEEE. His research interests are in CASE tools for both serial and parallel programs, and for software engineering in general.*

The Long Winding Road

Researchers have long been looking for alternatives to programming. These approaches can be roughly classified as follows:

- Linguistic — bigger and better high-level languages
- Systematic — elaborate systems for composing applications
- Prototyping — automatic generation of programs from "examples"



The linguistic approach has a rich history culminating in several contemporary camps: Ada in the military camp; object-oriented languages such as C++ in the maverick camp; and Prolog et al. dug in at the logic/AI/declarative-languages camp.

Small but significant productivity gains have been reported for Ada vs. Fortran applications, but there is still controversy over the productivity value of Ada. I will sidestep the controversy here because this is not my main point; suffice to say that the reported gains from programming in Ada are on the order of twofold. I am looking for techniques that reward programmers with a 1,000-fold increase because this is the quantum leap I believe will overcome the inertia of contemporary practice.

Object-oriented programming in languages such as C++ requires a shift in paradigm that will take a while to occur, but it is clear that this approach has advantages. Object-oriented programming can, for example, be combined with reusable-component technology to realize a fivefold increase in productivity after a suitable collection of reusable objects has been constructed.

The field of AI and logic programming has not yet produced measurable productivity gains, but the idea of a declarative programming style remains promising. In this style, a program is expressed as a declaration of facts and constraints rather than as a detailed prescription for how to solve the problem at hand. This is a powerful idea, and it, as well as the other approaches mentioned previously, has ramifications for prototyping.

All these languages are based on the idea that character strings contain enough expressive power to represent all that we want computers to do. In fact, the linguistic approach has been highly successful, leading to contemporary CASE tools based on data-flow diagrams and data-dictionary storage that are designed to work with these character-string languages. (Examples of these CASE tools are described elsewhere in this issue.)

Regardless, it would seem that the linguistic approach has run out of steam and something more powerful is needed. Instead of twofold or fivefold improvements, we need to invest in technologies that promise 1,000- to 1 million-fold improvements. To get such incredible leverage, we need to study methods of software production that transcend the written word.

The systematic approach is only about ten years old and is best represented by application generators and 4GL technology currently enjoying rapid acceptance

in the mainframe data-processing world. These systems combine many basic functions into a whole: screen forms generator, report generator, database query language, and processing function generation tools. Typical application generators reduce the time and effort to implement an application by a factor of 10 to 100, but they are restricted to narrow domains such as business data processing and database retrieval and reporting functions.

Prototyping is a novel technology that has been discussed for more than ten years, but little progress has actually been made until recently. The idea of a prototype occurred to software engineers as a method for capturing user requirements early in the life cycle so that the requirements could be examined, tested, and verified before actual coding began. Then, programmers realized they could automatically convert the prototype into running code for even greater productivity. Although a few prototyping systems exist, the advantages and limitations are not well understood. Prototyping, however, offers one of the most fascinating possibilities for vast increases in productivity. (See Figure 1, below.)

Before going on, it is only fair to mention an alternate approach to prototyping that is based on the linguistic paradigm. It is entirely possible to produce prototypes from high level specifications. Such specifications are normally written in a formal specification language. A few experimental systems for prototyping real-time control systems exist. These systems permit the direct execution of specifications to realize the prototype. Although interesting to the research community, executable specification languages are far too mathematically arcane to attract much practical use. I believe they will remain in the laboratory because they do not offer the possibility of giant gains in productivity.

Unfortunately, the major problem of software engineering — getting correct specifications, design, and implementation — remains difficult and costly largely because of the imperfection of telling

I will illustrate some rudimentary ideas of prototyping using an experimental system, called Oregon Speedcode Universe (OSU), currently under development by my research team. OSU is a software-development system employing on-screen editing of standard graphical user-interface objects, prototype sequencing, program generation, and a novel CASE tool for understanding source code. Programmers use OSU to design and implement all user-interface objects such as menus, windows, dialogs, and icons. These objects are then incorporated into an application-specific sequence that mimics the application during program development and performs the desired operations of the application during program execution.

OSU assumes a fundamental shift in paradigm: the user interface as language. This is a comfortable notion in the world of WYSIWYG and direct manipulation but requires a different perspective for the literate mind. In the world of WYSIWYG, programmers show what the computer is supposed to do instead of telling the computer what to do.

Showing Instead of Telling

Interest in visual programming, object-oriented design, and automated software-design methods has been heightened by the rise of graphical workstations that support windows, icons, menus, and pointing devices such as the mouse. The power of pictures over words has not been wasted on the users of these workstations, either. Rather, graphics-based computing has been combined with idealized models of the world — paradigms — to reduce learning curves, increase productivity, and generally remove the burden of program operation from the user. At the same time software developers have slowly come to realize that the user interface paradigm is itself a kind of programming language — a language that expresses the user's desires in pictures instead of words. This evolution can be characterized as a shift from "telling" to "showing."

Telling is performed by manuals, programming languages, and other written documents that attempt to convey instructions to user and machine alike. Telling involves two cognitive translations: from the idea to its textual representation and then back again from text to idea. To a software developer, these translations take place when a user's requirements are converted into code and then again when the code is executed. Unfortunately, the major problem of software engineering — getting correct specifications, design, and implementation — remains difficult and costly largely because of the imperfection of telling.

Showing is the process of doing in the

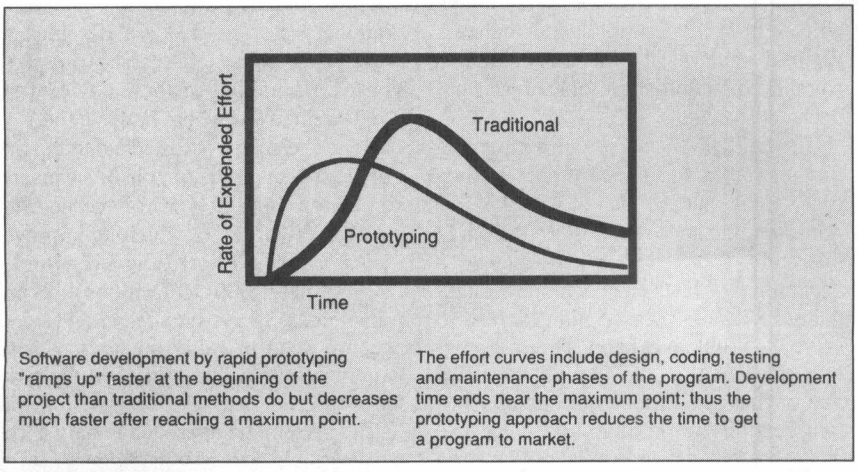


Figure 1: *Typical life cycle power curve showing how prototyping compares with traditional software development using languages and manual coding. The area under these curves quantifies the total effort in person-years, thus prototyping is not only faster but cheaper.*

(contined from page 20)
form of direct manipulation of "objects." Showing involves one level of cognitive translation: from the idea to its implementation. There is no linguistic ambiguity in showing because it is direct. Of course, humans train other humans by showing every day. Most people learn to drive a car from practicing rather than reading a book (often to the dismay of pedestrians, passengers, and innocent bystanders). Showing and doing are perhaps the most common forms of learning in the animal world.

Showing a computer what to do is difficult, and at present it is a less successful technology than traditional methods of giving instructions by telling via a programming language. But even an imperfect software tool for programming by showing can have dramatic impact on programming effort. Suppose, for example, that a certain application consists of 80 percent user-interface code and 20 percent calculation code. Further suppose that the 80 percent user interface code is automatically generated using a visual programming tool that captures what the user wants by showing. The effort to produce 80 percent of the code can be ignored, leaving only 20 percent (a fivefold increase in productivity) to be handcrafted by telling. By contemporary standards, any software engineering technology that delivers a fivefold leverage is considered revolutionary.

Great, but how do we program a computer by showing? Standard, graphical, user-interface-management systems based on a paradigm such as the metaphorical desktop provide a "platform" for showing vs. telling. The desktop metaphor of the Apple Macintosh is used here as a platform or prototyping language for expressing sequences of user-machine interactions. The particular user interface platform is not as important as the concept of "interface as language." In the remainder of this article, I'll show how a standard, consistent user-interface paradigm can be used to advantage in showing rather than telling.

One of the areas in which we win is in the ability to rapidly produce new applications that are easy to use

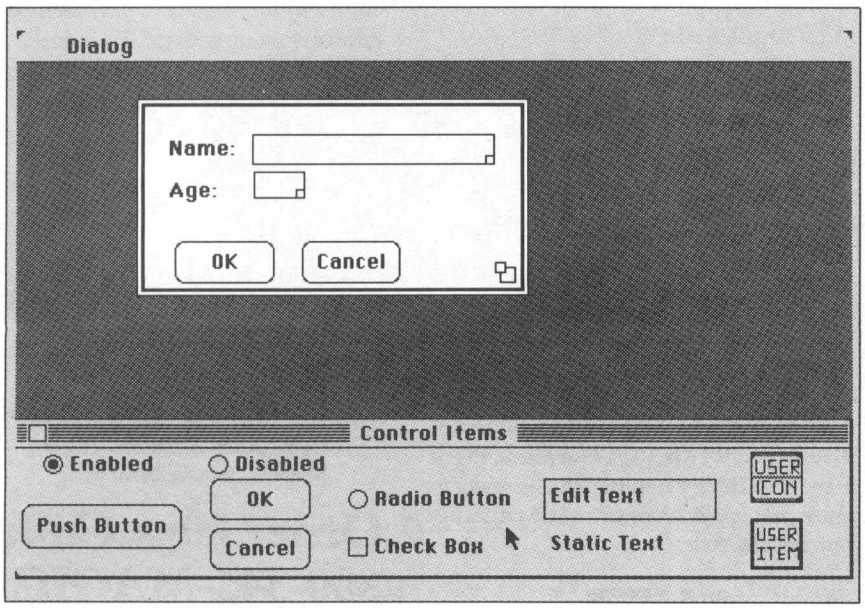


Figure 2: *A dialog contructed by dragging objects from the control panel and placing them in the dialog by pointing.*

The Theory of Prototyping

A prototype Q is a collection of user-interface objects U, a set of actions A, and a mapping function F, as follows:

$$Q = \{U, A, F\}$$

The user-interface objects U are the alphabet of symbols defined by a metaphor. The Macintosh desktop is a metaphorical desktop with an alphabet of icons for trashcan and files, pull-down menus, scrollable windows, and user-interaction dialogs. When used in a consistent manner, they form a language in much the same way that English characters form meaningful words when placed in strings according to the rules of English.

Construction rules for forming "words" in the desktop language can be expressed in English text, but there is a better way. Instead, user-interface objects can be constructed by direct manipulation of graphical "letters" such as menus, icons, and windows. An example of construction by direct manipulation is shown in Figure 2, below. In this example, an input form is constructed as a dialog containing standard letters from the alphabet. The letters are listed in the palette displayed below the dialog while it is being constructed. To insert an OK button in the dialog, simply drag one from the palette; to insert an editable text field into the dialog, drag a blank field from the palette and stretch it to any desired size. Similarly, radio items, check boxes, and icons can be placed wherever desired by doing, rather than by telling.

An immediate criticism of this approach is that the range of possibilities is limited by having a relatively small number of items to chose from. This is true, but it is also true that the expressiveness of a high-level language is limited by its alphabet and rules of program construction. Careful selection of such constraints is what software design is all about. For maximum flexibility, program in machine language. But if we desire reliability, quick development, and maintainability, we must carefully discard some powerful options in favor of more productive ones. When user interfaces are standardized, we lose some flexibility but gain in other areas. One of the areas in which we win is in the ability to rapidly produce new

applications that are easy to use. (As John Sculley says, "The programmer must give up control of the machine to the user.")

The actions *A* are the behaviors defined on the objects of the application program. We say the application is implemented according to principles of object-oriented design when two conditions are met:

1. Objects are encapsulated in clusters containing state and function — the state represents data, in general, and the functions define how to manipulate the objects.
2. The objects are manipulated exclusively by invocation of their functions — no state transitions are allowed by side effects of functions defined in any other encapsulation. (In a pure object-oriented system, the objects inherit their functions from a class, but I won't quibble about such details here.)

The objects in *U* that are seen by a user of the application are of immediate concern. These objects are activated by calling their functions. A menu is created and manipulated by function *GetNewMenu*, for example, and a window is displayed by its *GetNewWindow* function. In a standard user-interface management system, the behaviors for all user-interface objects are defined and fixed. They constitute the "verbs" in the "language" of prototyping; the state variables of each object constitute the "nouns."

Manipulating a member of *U* changes the internal state of an object. An open window is closed by calling its *CloseWindow* function, and a menu is disabled by calling its *DisableMenu* function. At any time, an interface is in a certain configuration — for example, one window is open, another is closed, a menu is disabled, and another is enabled. The sum total of the states of *U* constitute the configuration of the user interface.

The mapping function *F* is a graph describing state transitions from one user-*interface* configuration to another. State transitions in *F* are driven by the behaviors of the objects in the application program. We might, for example, want to show the computer how to display the dialog in Figure 2 by first selecting a menu item *OPEN*, followed by display of the dialog. This "simulation of the actual program" constitutes a change in the configuration of the user interface. The total collection of such changes in the user interface make up what we call *F*. Defining *F* is a challenging problem in practice. We look at the simple solution first and then explain the more difficult method of showing *F* to the computer.

F as in Finesse

In a mock-up, or vacuous prototype, the application interface is completely simulated but the application does nothing useful. None of the functionality of the application is carried out except the user interface. In terms of the theory, *U*, *A*, and *F* are defined but only for the portion of the application that interacts with the user. The dialog of Figure 2, for example, appears on the screen after the user selects the *OPEN* menu item and enters both *NAME* and *AGE* into the dialog, followed by clicking on the OK button. The application behaves as specified in the vacuous prototype, but the values of *NAME* and *AGE* are ignored!

For a standard interface, a vacuous prototype is constructed as follows:

1. Define all user-interface objects in *U*; inherit the standard user-interface object's behaviors as functions defined on each object.

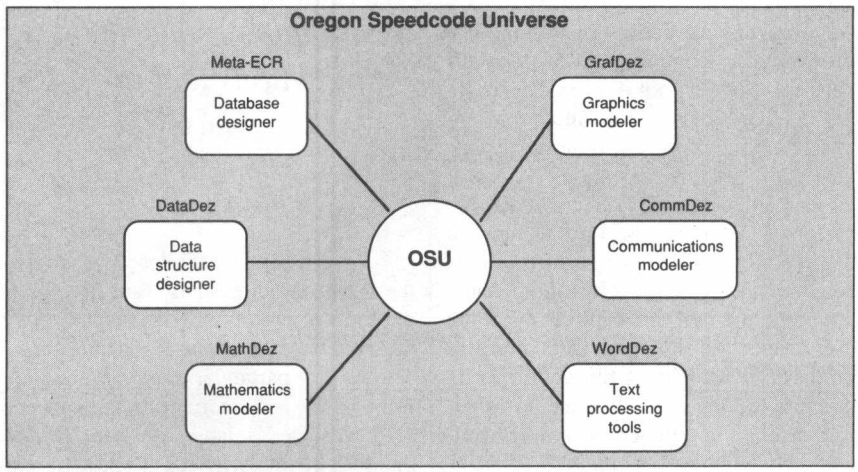


Figure 3: *Software Accelerators and the OSU Prototyper. Vacuous prototypes and user interfaces are designed in the central part of OSU. Software accelerators are stand-alone applications that generate "plug compatible" program parts for inclusion in the "action part" of a prototype.*

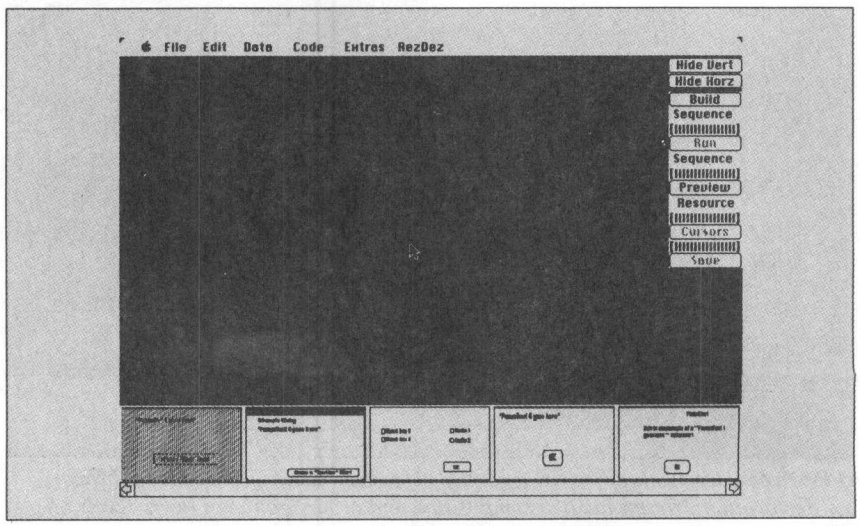


Figure 4: *The OSU desktop showing controls on the right, miniature user interface objects along the bottom screen, and the application's menus. A prototype is "sequenced" by selecting menus and user interface objects in the proper order.*

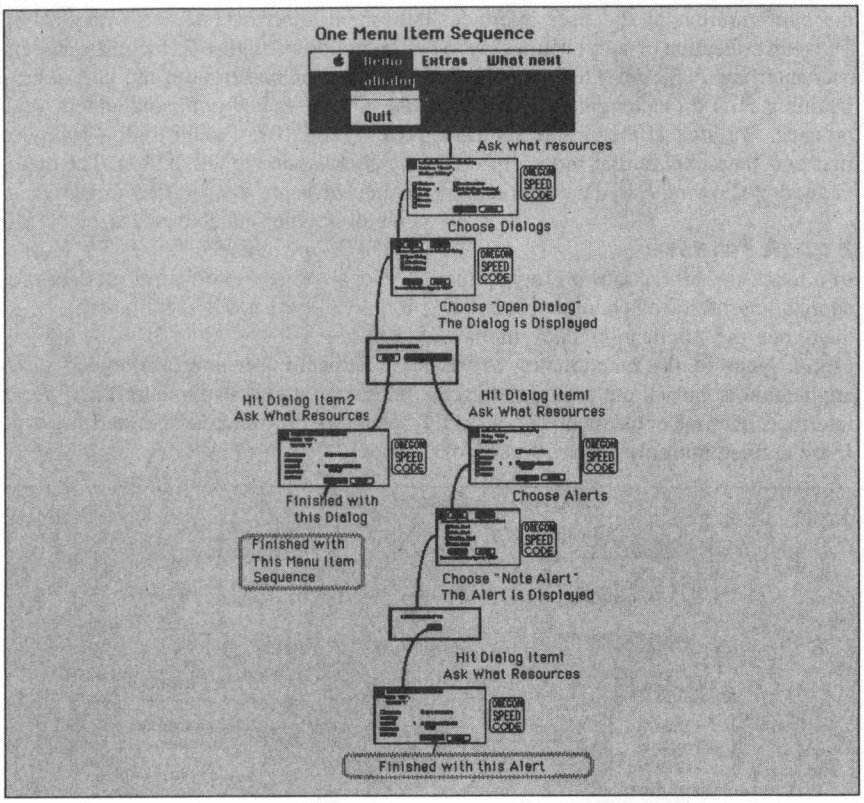


Figure 5: *A graphical sequence begins with a menu selection and leads to other user interface objects as shown here.*

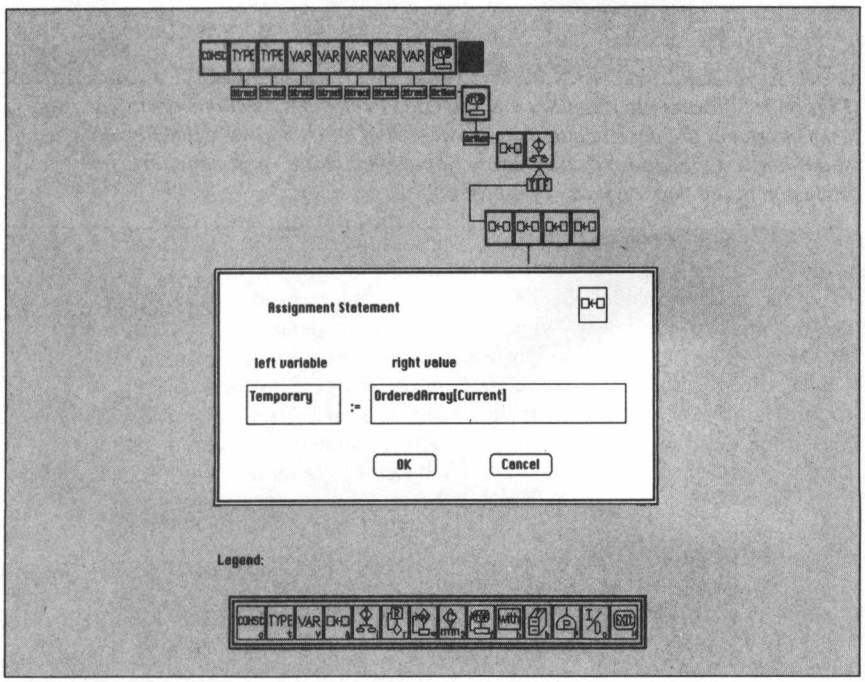


Figure 6: *Each icon represents a control or data construct such as loop, branch, or declaration. Hierarchical decomposition and/or navigation of lower levels of the detailed design of control and data constructs is achieved by selecting an icon with the mouse. In this example, we trace through four levels of decomposition beginning with the top I/O.*

2. Sequence the members of *U* by invoking the functions defined for objects in *U*. This gives rise to a set of configurations in *F*.
3. Generate code that implements *U*, *A*, and *F* as in steps 1 and 2.
4. Compile, link, and run the code produced in step 3.

Commercially available tools exist for creating vacuous prototypes — for example, Bricklin's Demo program for the IBM PC and SmethersBarnes Prototyper for the Macintosh. In addition, application-development systems incorporated into database-management systems such as dBase III and 4th Dimension employ similar tools for creating custom user interfaces to match the application.

Creating vacuous prototypes is relatively easy, but creating full-fledged prototypes for realistic applications is within current technology only when the domain of application is severely restricted. Domain-specific prototyping systems exist for certain real-time control problems and certain classes of business applications, for example. (I might claim that a spreadsheet is a domain-specific prototyping tool, but it might evoke too much negative correspondence.)

Prototyping systems for general applications — applications that can currently be implemented by telling rather than showing — are beyond current technology. We claim that wide-spectrum prototyping systems will be achieved in the near term by combining several domain-specific tools. When domain-specific tools are combined with vacuous prototyping, the result is a wide-spectrum prototyping system. Such systems are still experimental, but when they are perfected, programming will be more like flying an F-14 than writing a term paper for English Lit.

Prototyping at the Edge

OSU is a program-development system based on the notion of a wide-spectrum prototyper. It incorporates several domain-specific tools for automatically creating, manipulating, and "playing back" prototypes. In addition, OSU incorporates some CASE-like features for doing traditional coding, thus retaining the power and flexibility of traditional high-level-language programming. Complete applications are generated from OSU proto-

(continued from page 24)
types — currently in the form of compiled and linked LightSpeed Pascal programs.

The core of OSU consists of four tools for graphically constructing $Q=\{U, A, F\}$ — ResDez; a graphical sequencer; a program generator; and Vigram, a detailed design and program-comprehension tool.

ResDez (Resource Designer) is used to create and edit all user-interface objects graphically — menus, icons, dialogs, windows, alerts, error messages, prompts, and associated information. These objects are "painted" on the screen exactly as they initially appear in the finished application. Therefore, ResDez not only creates each object but it also defines the initial internal state of the object. (See Figure 3, page 23.)

A second tool, called a graphical sequencer, is used to create *A* and F — all configurations and the actions for transforming elements of *U* from one state to another. The graphical sequencer is used by a programmer to "play out" the application by doing rather than writing instructions in the form of a script or textual language. The actions of *A* are the behaviors of the desktop objects defined by the standard user interface.

A screen dump of the sequencer in action is shown in Figure 4, page 23. The application's menus are in the menu bar, and miniatures of the user-interface objects are shown along the bottom of the screen. Actions are shown to the application by pointing and clicking, just as you would in the actual application. The sequencer is itself operated from the buttons on the right side of the screen in Figure 4.

A prototype can be created, sequenced, and played back like a movie. Each action is initiated by actually doing it, but often additional information is needed to clarify the action. When needed, additional information is obtained through OSU dialogs that ask for details such as the state of a menu item after it is selected (checked, disabled, and so forth) or the disposal of a dialog. Figure 5, page 24, shows a sequence created within OSU.

The third tool is a program generator that automatically writes compilable Pascal source code equivalent to the prototype defined by *U* and *F*. Several alternative methods of code generation might have been employed in OSU: direct compilation, translation into intermediate code, and direct interpretation. We chose source-code generation because it takes advantage of compiler code optimization, gives the programmer access to a maintainable version of the program, and the resulting prototypes are easily combined with other program components taken from libraries and other languages.

To make prototyping interesting to a wider audience of programmers, OSU must cover a wider array of applications

Given a graphical sequence, the code generator writes a program that carries out the steps described by the sequence. The code generator works behind the scenes and is not part of the user interface of OSU.

Finally, Vigram (VIsual proGRAMming), is a tool for constructing graphical views of the source-code text generated by the code generator and/or by hand-coding the old-fashioned way. Vigram serves two purposes: detailed design of individual procedures and automatic construction of a detailed design schematic of an existing procedure so that it can be understood at a glance. Figure 6, page 24, shows a Vigram "picture" of a Pascal procedure.

As you can see, OSU does not try to do everything automatically. A large part of the coding task is automated only if a program can be constructed from standard user-interface parts and standard program functions. But if a particular algorithm or technique must be hand-coded, OSU permits an escape by way of "Vigramming." Yet productivity gains are minor when you compare Vigramming with traditional text editing. The real advantage of Vigram is in comprehending existing programs for the purpose of reusing them. An existing program is entered into Vigram, turned into a picture, studied, and adapted to its new purpose. Thus, Vigram is a tool for adapting reusable components.

The core of OSU permits the construction of standard data-processing applications. These programs are controlled by a system of menus, data-entry dialogs, and limited amounts of graphics. They do not contain novel algorithms for word processing, interactive graphics, sound, telecommunications, and so on. Of course, these are exactly the kinds of things we want to do with computers, though! To make prototyping interesting to a wider audience of programmers, OSU must cover a wider array of applications.

A wide-spectrum prototyping system must be flexible enough to generate applications in many problem domains. The system must be able to prototype and generate applications in document processing, interactive graphics, sound, telecommunications, and a diversity of data-acquisition and control-applications. To do this, we need a large family of domain-dependent tools called software accelerators.

Software Accelerators

Software accelerators accept direct manipulation of various objects as inputs and produce object-oriented-code modules as output. Here *object-oriented* means that data and the operations that can be performed on the data are encapsulated in the form of a Pascal unit. Every Pascal unit contains an interface part that defines the constants, types, and procedures for operating on the encapsulated data structure. These units are automatically generated by showing rather than telling. The units are then plugged into the prototype using their interface parts as the "plugs." Briefly, the tools, which are shown in Figure 3 are:

- Meta-ECR: A design tool for designing and creating databases modeled after the entity-category-relation model
- DataDez: A design tool for designing and creating internal data structures and the algorithms that operate on them
- MathDez: A design tool for mathematical modeling and describing calculations

by direct manipulation
- WordDez: A design tool for text editing and processing
- CommDez: A design tool for generating communications algorithms
- GraphDez: A design tool for generating algorithms to manipulate graphical objects in an application

I do not have space to explain each of these tools in detail.

The uniformity of code interfaces across all objects in the system enable OSU to incorporate functionality from any domain-specific tool. Thus, OSU is not limited to these tools because of the uniformity of the code interface. Additional domain-specific tools can be incorporated into OSU as long as they conform to the code-interface restrictions placed on the object-oriented units. In this way, we can grow from a limited, vacuous prototyping system to a robust, wide-spectrum prototyping environment in which programmers realize orders of magnitude of improvement in productivity. But then, this is the subject of another article!

Conclusion

The goal of CASE is to improve productivity in all phases of the software life cycle — planning, design, coding, testing, maintenance, and management. Prototyping is a radically different approach that tries to eliminate rather than support the phases of the life cycle. Prototyping compresses design, coding, testing, and maintenance into a single step. Modifications to an existing system are made simply by regenerating the complete application because generation is totally automated.

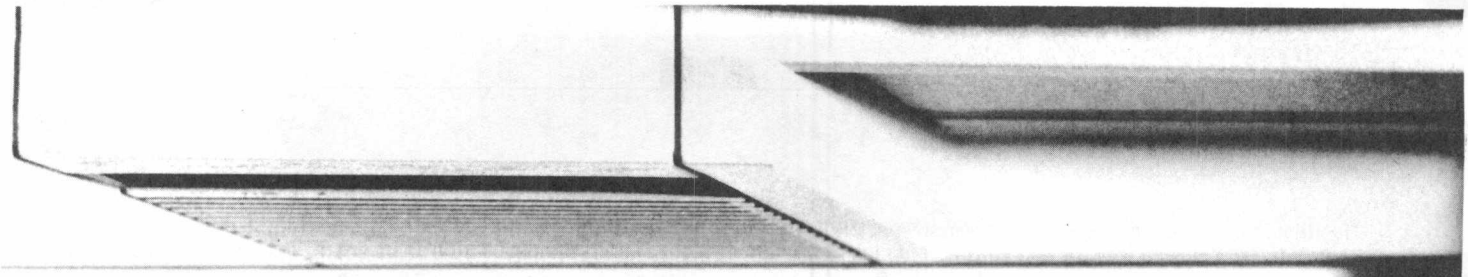
These notions are difficult to accept because they come from a different perspective — a paradigm shift is prerequisite to appreciation of prototyping. Caution is in order, however, because prototyping is an immature technology. Much more fundamental research is needed to make it a practical reality.

Acknowledgments

OSU is a long-term research project designed and implemented by a large number of students, including the following outstanding members of the research team: Fred Handloser III, Sharada Bose, Sherry Yang, Chi-Chia Hsieh, Kritawan Kruatrachue, Jagannath Raghu, and Jim Armstrong. Details on these projects are available as technical reports from the Computer Science Department, Oregon State University, Corvallis, OR 97331; 503-754-3273 (ask for Pat).

DDJ

Vote for you favorite feature/article.
Circle Reader Service **No. 2.**



Using an API as a Developer Platform

Increase programmers' productivity by using an application-specific system

by Jeffrey M. Parker

Modern computer systems consist of underlying hardware with several layers of operating systems, virtual machines of different architectures, programming systems, and finally, "applications." The role of the systems software engineer is to produce higher-level layers for use by the applications software engineer, who adds the outermost application layer. Part of the ubiquitous software production bottleneck is because applications software engineers too often are working with programming systems that are too primitive and too far removed from their application areas. Dealing with the idiosyncrasies of the end user and of applications domain is enough of a challenge by itself. Applications software engineering shouldn't be complicated by mapping the application into an unsuitable programming system for the lack of something better.

At one time, the question of advanced programming systems was moot. Memory limitations, low CPU performance, crude display technology, and so forth, dictated that a programming system required as few resources as possible. Today, that computer hardware and software technology have evolved to the point where blazingly fast CPUs, megabytes of memory, and dazzling graphics are available even on relatively low-cost personal computers. Today, more resources can be devoted to higher-level, easy-to-use programming systems.

The properties of a programming system germane to the task of applications software production are as follows: the computation model; the programming metaphor and tools; and the basic program building blocks of the virtual machine upon which the application is to be built. Typically, a programming system is built as an extension of the underlying machine. The computation model, programming metaphor, and building blocks are chosen with more emphasis on how well they fit on top of the machine, rather than how well they support a given application area.

An alternative approach is to focus on a particular application area. This produces a software-development system whose computational model, programming metaphor, and primitive building blocks are tailored to that application area. This approach leverages the efforts of the applications software programmer, as shown in Figure 1, page 30.

Consider the computer-aided test-and-measurement application area. In this area, computers are used as tools for controlling and collecting data from electronic instruments and from plug-in data acquisition cards. In this area, computers are used for processing and displaying the data in various ways.

The test-and-measurement domain has some peculiarities that make test-and-measurement programming different from, say, writing an operating system or an accounting package. Test setups change frequently and software needs to be easily modifiable. A need exists for low-level software components (such as hardware drivers) as well as for some high-level components (such as the ability to graph data easily).

Test-and-measurement programs are usually not written by professional programmers. Instead, a scientist, engineer, or technician — an instrumentation user —does the programming, and these people don't necessarily think the way professional programmers do. Concepts such as control settings, block diagrams of test setups, and signals flowing between instruments are familiar and natural to the instrumentation user. Concepts like variables, designing in pseudocode, and sequentially executed program statements may be foreign to them.

LabVIEW, from National Instruments in Austin, Texas, is an application-specific programming system that runs on the Macintosh. Its computation model, programming metaphor, and program build-

Jeffrey Parker is a software design engineer at National Instruments Inc., 12109 Technology Blvd., Austin, TX 78727-6204. He is working on open architecture and advanced data display features for LabVIEW.

ing blocks were chosen with the test-and-measurement application area in mind. The remainder of this article makes use of the test-and-measurement application area and of LabVIEW, as examples will illustrate. The idea of creating a higher-level programming system by tailoring its properties — computation model, programming metaphor, and building blocks — to fit a particular application area.

Computation Models — Control Flow and its Alternatives

The computation model defines how computation proceeds in a computer. The most prevalent computation model is the control-flow model. Here, the computer's instructions and data reside in global memory and instructions are executed under the control of a program counter. Obviously, many applications can be mapped onto the control flow model, but that mapping isn't always natural. The easiest programming to build is one that mimics the underlying hardware, and that's the main reason for the popularity of the control-flow model. The von Neumann global memory-centralized processor architecture is still the most prevalent hardware architecture.

Many alternative models of computation are better suited to certain application domains. Two well-known examples are the functional model of computation and logic programming, both of which were developed and refined to satisfy the needs of the AI domain.

Hardware support is becoming available for many computation models. The benefits of a different computation model to an application area are often so great that it pays to build the new computation model in software on top of hardware based on a different model. This is true in the test-and-measurement area. The concept of signals flowing through functional blocks that perform transformations on them is familiar to test-and-measurement users. A natural extension is to think of data as flowing through a computer program. For this reason, LabVIEW's computation model is a virtual dataflow machine — virtual in the sense that it is implemented on top of the von Neumann architecture of the Macintosh 680xx processor. A dataflow machine does not rely on a central processor sequentially executing instructions that manipulate global data under the control of a program counter. Rather, a dataflow machine executes instructions when their data is available. When an instruction has all of its data inputs, it can execute (or "fire") and produce a result at its output(s), which may then allow another instruction that is waiting for that output to fire, and so on. At a given time, if multiple instructions have all of their input data available, they can fire in arbitrary order. If more than one processor is available, multiple instructions can make use of the additional processors to run at the same time.

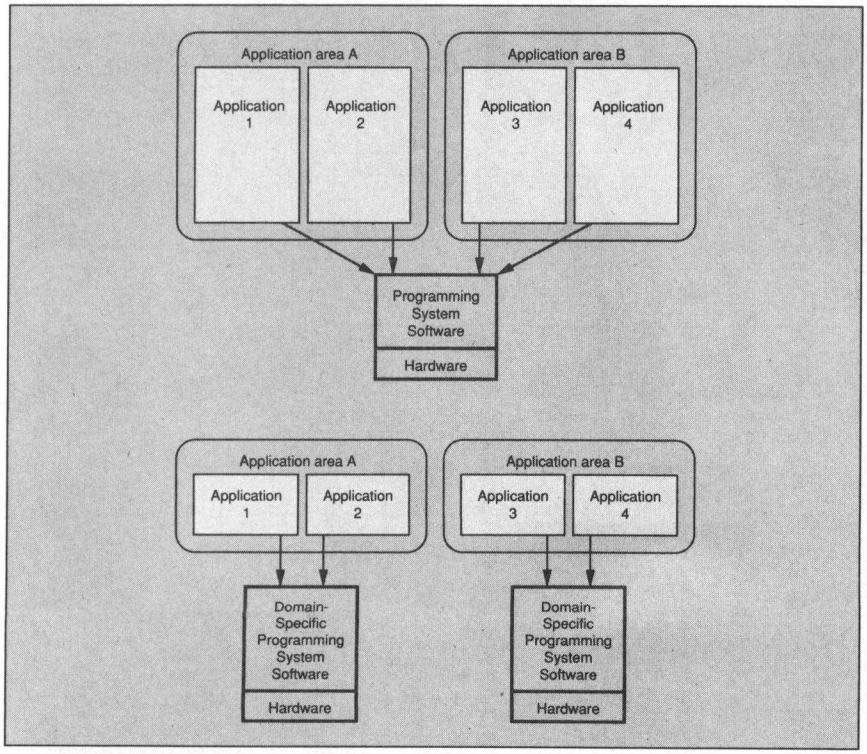


Figure 1: *The top diagram shows several applications built on top of a low-level programming system. The programming system software and hardware make up the virtual machine upon which the applications software is built. The amount of work done by the applications programmer is represented by the shaded boxes. The bottom diagram illustrates the leverage provided by tailoring the programming system to fit the application domain, instead of the underlying hardware.*

Programming Metaphors: Text Languages vs. Graphical Programming

The programming metaphor determines what form a program takes, how its data is defined, and how the program is to be constructed. In other words, the programmer metaphor determines how the user manipulates the computation model.

Most computation models are still using the programming metaphor that was introduced in the 1960s: the programming language based on a context-free grammar with textual tokens. Text-based programming languages have many strong points. These languages are a concise representation of abstract programming concepts, and can be made unambiguous. They're adaptable to a wide variety of underlying virtual machine architectures, and developing tools to work with them is easy. Text languages are flexible and portable. These advantages ensure that the text-based programming languages will be with us for a long time.

Text-based programming metaphors have some disadvantages, as well. Note that most of the advantages just mentioned are due to the ease with which a text-based programming metaphor can be built *on top of* the underlying computational model. When starting from the perspective of the application domain and looking inward to the computation model, a new set of criteria becomes more important. How well does the programming metaphor *fit underneath*, and relate to,

that application domain? From this viewpoint, the shortcomings of a text-based metaphor become clear.

A major shortcoming of text-based programming metaphors is the difference in level of abstraction from the problem to be solved in the application domain. These metaphors may be a good specification of the computation machine on which the application will execute, but they're usually not a good description of the application itself. The application programmer who uses these metaphors must abruptly "shift gears" between thinking about the application in terms of the problem domain and in terms of implementation on the computation model.

Text-based programming languages are usually not much help in the early stages of designing a program. Most software designs begin (or should begin) with figures and diagrams of various kinds — dataflow diagrams, state diagrams, flowcharts, and so on. The human mind is visually oriented. It captures and understands complex relationships more quickly when they are presented pictorially. Consider a simple problem from the test-and-measurement domain. An engineer wants to use an analog-to-digital converter to acquire some data, scale it to a different range, and plot it on a graph. The design would probably be worked out in the form of a block diagram, such as the one in Figure 2, this page.

Given this design, the first step in writing a program to implement it in a programming language such as C might be to recast the block diagram in high-level pseudocode:

```
BEGIN
AcquireData(D);
ScaleData(D, K1, K2);
AddTimebase(D, X0, FX);
PlotGraph(D);
END
```

Even at this relatively high level of abstraction, the ability to see what the program is doing has been lost. While mapping the pseudo code into compileable C code, the programmer becomes further immersed in the syntax and semantics of a programming metaphor that has more in common with the machine than the application. Relationships that were explicit in the diagram become implicit in the text code. For example, the fact that the data array is being transformed by successive operations is obvious from the diagram, but hidden in the pseudocode. Based on data dependency, the order of execution is clear from the diagram, but only implied in the pseudocode. And such problems are amplified as the four lines of pseudocode are expanded into tens or hundreds of lines of C.

The drawbacks of the dichotomy between design process and program implementation are compounded when, changes in the program's design must be made during the development or maintenance of an application. When the application design process is on a different conceptual plane than the program implementation, all design iterations must make the long round trip through implementation at a different level of abstraction. This is particularly troublesome when the nature of the application itself calls for a rapid-prototyping development style and, therefore, significant iteration at the design level.

To illustrate, consider the test engineer who wants to plot the data both before and after scaling in the data acquisition example — a seemingly trivial change. Modifying the block diagram is easy enough (as shown in Figure 3, below). To change the C language implementation, the programmer must shift gears and delve back into a different model. It is important to recognize, for example, that D is a shared storage location. The order of computation is important. If the plotting function as a side effect modifies D, then a copy of D must be made. Uncertainty about the side effects of what appear to be simple changes in the design may result in unreasonable resistance to beneficial changes.

Yet another drawback of conventional

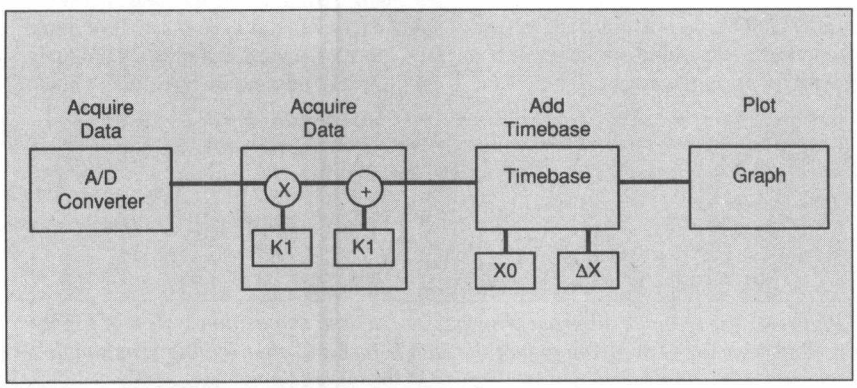


Figure 2: *Block diagram of a simple test-and-measurement system*

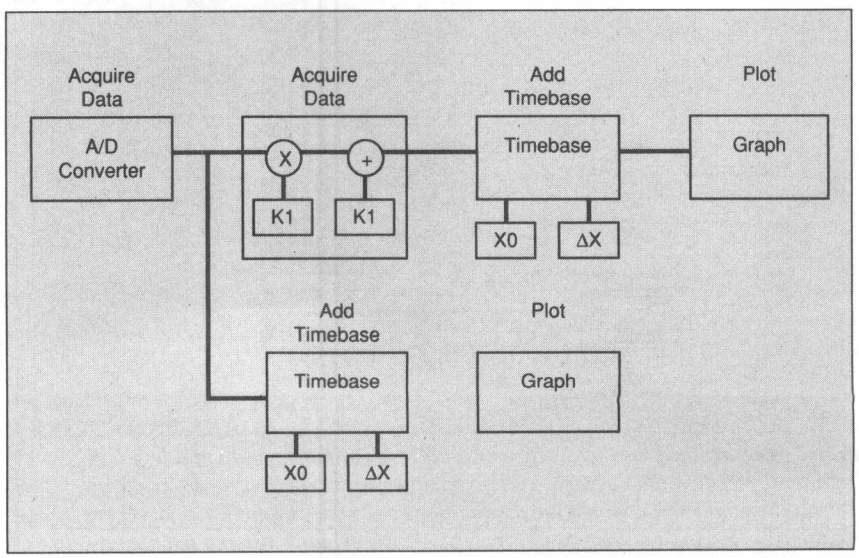


Figure 3: *Block diagram of the test-and-measurement system of Figure 2, modified to plot graphs before and after scaling*

text programming languages is that source code is usually poor documentation of the program's design. Other internal documentation must be generated, if the program is to be maintained.

Since diagrams are often a better representation of an application and its design process, why not produce executable code directly from some sort of diagram? That is the idea behind graphical programming metaphors. Using a graphical editor (the counterpart of a text editor), the programmer draws a diagram which, if syntactically correct, can be directly executed. A graphical programming metaphor that is carefully chosen to fit the application area can greatly leverage the efforts of the application software engineer. This is accomplished by minimizing the abruptness of the transition between the design of the solution and its implementation as a program.

Programming by Diagram

In LabVIEW, the programming metaphor takes the form of a graphical dataflow programming language called G. In LabVIEW terminology, a G graphical program is referred to as a "block diagram." This metaphor was chosen for two reasons: block diagrams are a familiar concept in the test-and-measurement world; and dataflow programs are well-suited to a graphical representation, that is, nodes or instructions, connected by arcs or data paths. The primitives of G consist of icons that represent sources and sinks for data (called "terminals"), and icons that represent "instructions" (or units of execution). The LabVIEW programmer constructs a block diagram by connecting data outputs on the primitive icons to data inputs on other primitive icons.

Figure 4 left, illustrates the G program for the data-acquisition example. Note the similarity to the block diagram of Figure 2, where data conceptually flows along the lines connecting the various blocks. (Also, note the ease with which the graphical program can be modified to produce the system of Figure 3). In the LabVIEW block diagram, data flows between dataflow instructions. These instructions vary in complexity from simple (addition and multiplication) to complex (acquire an array of data, add a timebase for plotting). The larger block in the center of the LabVIEW program (the one that resembles a pad of paper) is an iteration structure called a *for loop*. It iterates over the elements of the array, performing the scaling operations on each element. The interior of the *for loop* is a dataflow sub-graph that executes once per iteration of the loop. The result array is accumulated over the iterations and becomes the output of the *for loop*.

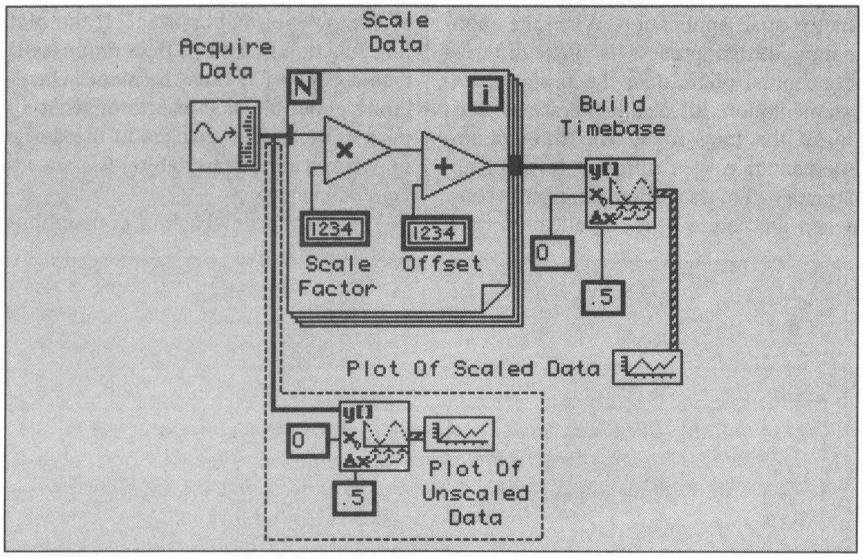


Figure 4: *The G block diagram program for the data acquisition system of Figure 2. Modifying the G program to plot the data before and after scaling is as simple as adding the program elements surrounded by dashed lines.*

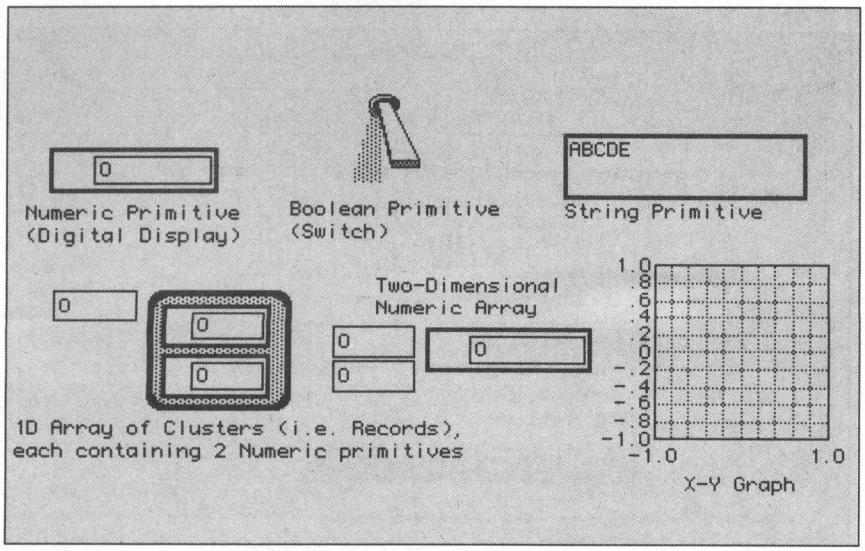


Figure 5: *LabVIEW's basic and aggregate control styles. Arrays have one auxiliary digital display per element for selecting the element to be displayed. The X-Y graph is an alternative display style for the array of clusters at the lower left.*

Data Declaration: the Front Panel

The test-and-measurement programmer defines data in LabVIEW by using another familiar metaphor: the front panel of an electronic instrument. Data resides in objects called "controls" that the LabVIEW programmer arranges in a window called the "front panel." The kind of control determines the type of the data it contains, just as with physical instruments (toggle switches represent Boolean data and digital readouts represent numeric data). LabVIEW also has a string primitive data type, and a corresponding front-panel control for entering and displaying text. Aggregate data types —arrays and

structures ("clusters") — can also be represented. A special display style, the graph, displays a special case of aggregate data type (an array of X-Y points). Figure 5, page 31, shows LabVIEW's basic and aggregate control styles.

The method for entering data values is consistent with the control metaphor. Values are entered by "operating" the control (the mouse or the keyboard). Thus, Boolean switches are "flipped" by clicking on them with the mouse, and the value in a digital display is changed by typing in its new value.

Front Panel + Block Diagram = Virtual Instrument

Another important metaphor in the LabVIEW programming system concerns the relationship between the front panel window (where data is specified, entered, and displayed) and the block diagram window (where the G program is created). Figure 6, right, shows the front panel window and block diagram window for the data acquisition example. Each control or indicator in the front panel window has a corresponding terminal icon on the block diagram, so that its data may be introduced into the G program. In Figure 6, the controls and indicators in the front panel window labelled "Scale Factor," "Offset," and "Plot Of Scaled Data" correspond to the similarly labelled terminals in the block diagram window.

This is analogous to a real electronic instrument. If you look at its controls and indicators from the front-panel side, you are seeing its user interface. If you remove the outer case of the instrument and look at the front panel from the back side, you see the terminals that allow the controls and indicators to be connected to the circuit.

Together, a front panel user interface and a "circuit card" in the form of a G-language block diagram program make up LabVIEW's basic module, which is called a "virtual instrument."

LabVIEW's Graphical Editor and Execution Executive

Graphical programs are constructed with a graphical editor. A good metaphor is just as important for this programming tool as it is in the graphical programming language itself. LabVIEW's graphical editor extends the electronic instrument metaphor where appropriate, and draws on the metaphors of the Macintosh as well. The standard Macintosh window and menu interface is employed, as is the idea of direct manipulation of graphical objects made popular by the Macintosh Finder's desktop metaphor.

The LabVIEW programmer builds front panels and block diagrams by choosing objects (such as controls and dataflow instructions) from menus, which causes them to appear as icons in the appropriate window. They can then be copied, deleted, and positioned as desired by using the usual Macintosh pointing, clicking, and dragging mouse operations. Pulldown

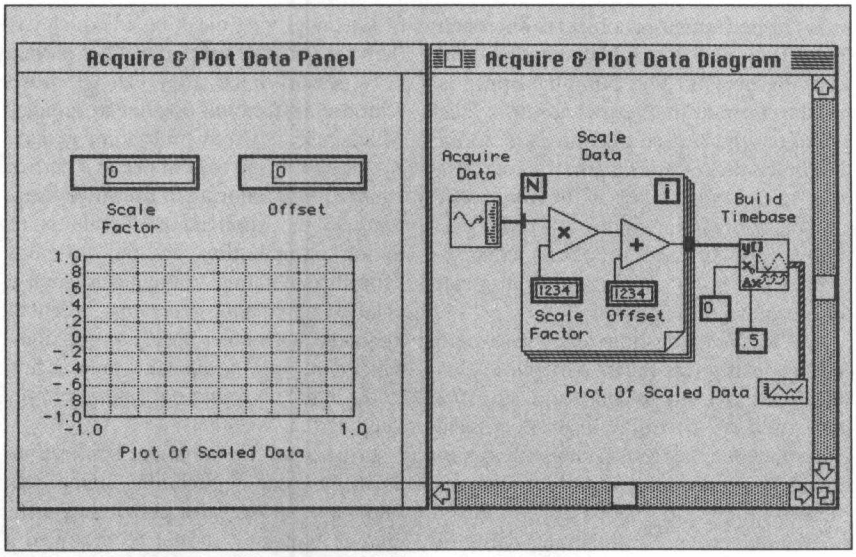


Figure 6: LabVIEW's basic module, the virtual instrument, consists of a front panel interface combined with a block diagram graphical program. Each control and indicator on the front panel has a corresponding terminal on the block diagram.

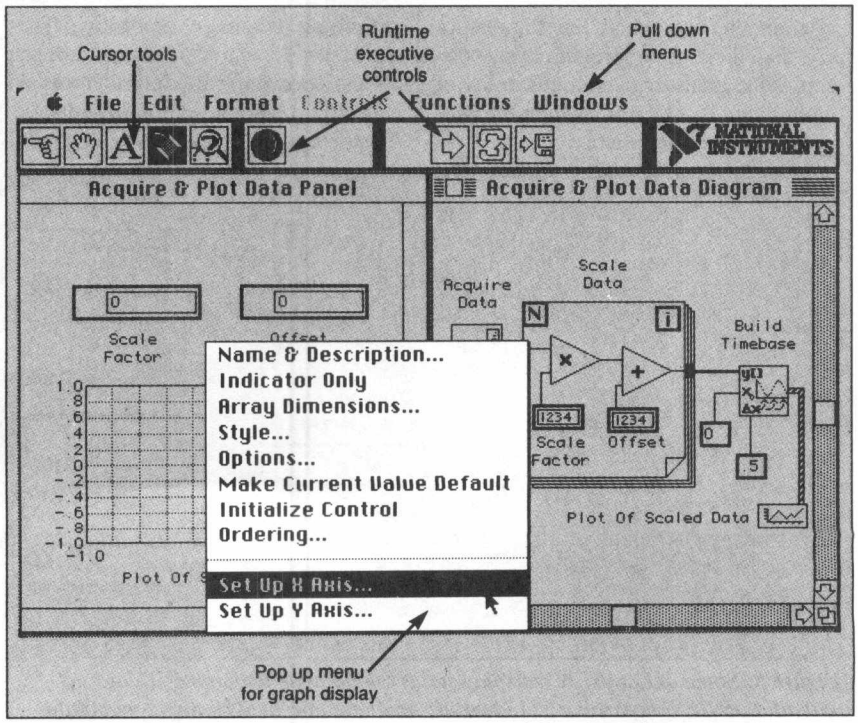


Figure 7: Features of LabVIEW's graphical editor and execution executive

menus are generally reserved for options that are more global in nature. Attributes of objects are accessed by using popup menus that supplement the pulldown menus. Popup menus in LabVIEW are context-sensitive (that is the menu is specific to the object that is popped up on).

The cursor takes on the shape of various "tools" to indicate what operations may be performed on objects. The open hand is used for moving, resizing, and copying objects. The pointing finger is used to operate front-panel controls. The spool of wire is used to connect icons on the block diagram. The I-beam cursor is used to create text labels in the front panel and block diagram windows. The magnifying glass is used to obtain context-sensitive help by clicking on front panel or block diagram objects.

LabVIEW's run-time executive also has an instrument panel metaphor. This metaphor is used to start and stop the execution of a virtual instrument, and control debugging functions (such as single-stepping through dataflow instructions).

Figure 7, page 33, illustrates some of the features of LabVIEW's graphical editor and run-time executive.

Program Building Blocks

In addition to the computation model and programming metaphor, the final property that pertains to programming systems is the granularity and reusability of the building blocks from which the programmer creates programs.

How Primitive Are the Program Primitives?

The granularity of program building blocks is determined by the computation model and programming metaphor, which, when combined, provide the programmer with a set of primitive constructs. Like the programming metaphor itself, these primitives can be tailored to fit the underlying machine, in which case they are general, but perhaps not particularly powerful. Or, they can be chosen with the application domain in mind, in which case they may trade some generality for power and ease of use in a particular area. This tailoring of primitive operations to the application domain is the major contribution of the so-called "fourth generation" languages (such as database application generators, where the language's primitive operations do in one statement what might take several lines to do in a "conventional" programming language).

In test-and-measurement applications, as in database applications, some complex operations are performed often enough that they should be provided as primitives by the programming system. LabVIEW, like fourth-generation languages, supplies some higher-level constructs, in addition to the primitives that give LabVIEW its general-purpose programming language flexibility. These higher-level constructs fall into three categories: special front-panel controls and indicators, higher-level primitive dataflow "instructions," and libraries of prebuilt virtual instruments.

Test-and-measurement applications programmers need more flexibility in front-panel data input and display than is offered by the basic control and indicator styles of Figure 5. This is for a variety of reasons:

- Engineers often want to produce a front panel that looks as much like a physical instrument as possible. That way, users who are familiar with the physical instrument are also immediately familiar with its virtual instrument representation.
- Different control styles help the user to distinguish quickly among several controls on a crowded front panel.
- Analog numeric displays have some advantages over digital displays. They can impart rate-of-change information as well as instantaneous value, and they have the ability to provide at a glance a relative measurement (for example, "the tank is half full").

Figure 8 this page, illustrates several of the display styles for Boolean and numeric data provided by LabVIEW as system primitives.

LabVIEW provides a wide variety of primitive dataflow instructions for manipulating data of all types. Some of these primitives are identical to what you would expect to find in any programming language. Basic arithmetic operations (such as add, subtract, multiply, and divide), Boolean operations (AND, OR, NOT), and basic string operations (such as concatenation) are provided. Also provided in LabVIEW's toolkit are primitives for communicating with hardware through internal ports and busses, and specialized string functions that are useful for building and parsing strings for command-based automated test instruments.

Finally, several libraries of virtual instruments are produced in-house and distributed with LabVIEW because they satisfy special needs within the test-and-measurement domain. A high-level mathematical library provides virtual instruments for digital signal processing and statistics. A library of virtual instrument drivers provides access to National Instruments' data acquisition plug-in cards. A variety of instrument drivers provide a virtual instrument layer on top of the usual command-based remote programming interface of automated test instruments.

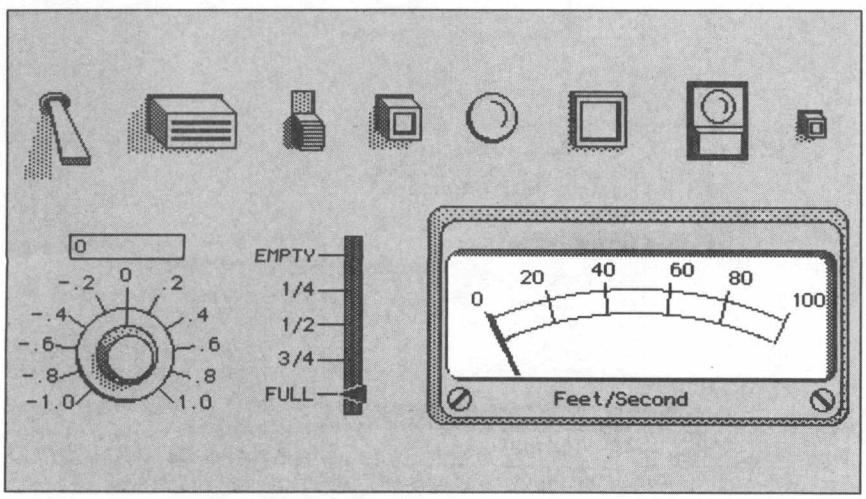


Figure 8: *Some of LabVIEW's display styles for Boolean and numeric data. As with all LabVIEW controls and indicators, they are operated by direct manipulation (that is, by "dragging" the moving part around with the cursor).*

Reusability: Why Reinvent The Wheel?

Unlike other disciplines, in present-day software engineering, products built from reusable components are the exception rather than the rule. Reusability depends on several factors. Does the programming metaphor facilitate the construction of hierarchical, reusable components? Are *the* components packaged so they can be easily reused? Finally, what about cataloging and documentation? How difficult is it to find something to reuse and given a component, how difficult is it to figure out what it does? Most programming systems provide a way to construct modules, and some also provide good packaging and browsing capabilities (for example, the object-browsing capabilities of Smalltalk). In most systems, software reusability suffers because of packaging and documentation problems.

LabVIEW addresses the software reusability issue by allowing its basic modules (virtual instruments) to be encapsulated and used as dataflow instructions in the block diagram programs of other virtual instruments.

Three steps are involved in encapsulating a virtual instrument (see Figure 9 this page). First, the virtual instrument must be given a graphical identifier (an icon that is created with a builtin icon editor). Next, its front panel interface must be supplemented with a programmatic interface that allows it to be connected into another block diagram program. Since the controls and indicators on the front panel already define the data interface to the block diagram program, they are associated with areas on the icon to define the programmatic interface. Finally, the virtual instrument must be stored in a way that allows it to be retrieved for use in another block diagram program. Each virtual instrument is stored in a separate file and retrieved by name.

From the standpoint of software reusability, virtual instruments are excellent modules. The concept of a "software integrated circuit" has received a lot of press, but not a lot of practice. LabVIEW's virtual instruments are quite literally software integrated circuits, with a well-defined "pinout" for connecting them into a graphical program. The storage granularity of a virtual instrument (one per file) simplifies the logistics of reusability. No library manager is needed. A virtual instrument carries its source code, user interface, and documentation in one package. The LabVIEW programmer can open the virtual instrument's block diagram to see how it works, try it out by operating it interactively from its front panel, and then use it programatically as a submodule in another virtual instrument's block diagram.

Conclusion

Are high-level, domain-specific programming environments effective? Judging from the success of LabVIEW and other products that fit the high-level, domain-specific model (such as the 4th Dimension and Helix databases, Visual Interactive Programming, and HyperCard), the answer is "yes." Since LabVIEW was released as a product in the fall of 1986, it has gathered an enthusiastic following. Its users have reported that they are, indeed, able to more rapidly build test-and-measurement applications because of LabVIEW's advanced programming metaphor and built-in components.

Two years of feedback from LabVIEW users has revealed what users need and expect in a product like LabVIEW. Much of what has been learned (especially in the area of execution performance and editing functionality) has been incorporated into Version 2.0.

From the developer's point of view, advanced programming systems probably look like a lot of work to implement — and they are.

The fact is, the systems software engineer *does* have to spend some of that memory, and some of those processor cycles, and quite a bit of his or her time, to make the benefits of easier-to-use programming systems available. Advances in hardware technology mean more resources are available to spend. The key concept here is that they're being *spent*, not wasted. With high-level programming systems comes leveraged increase in software engineering productivity.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

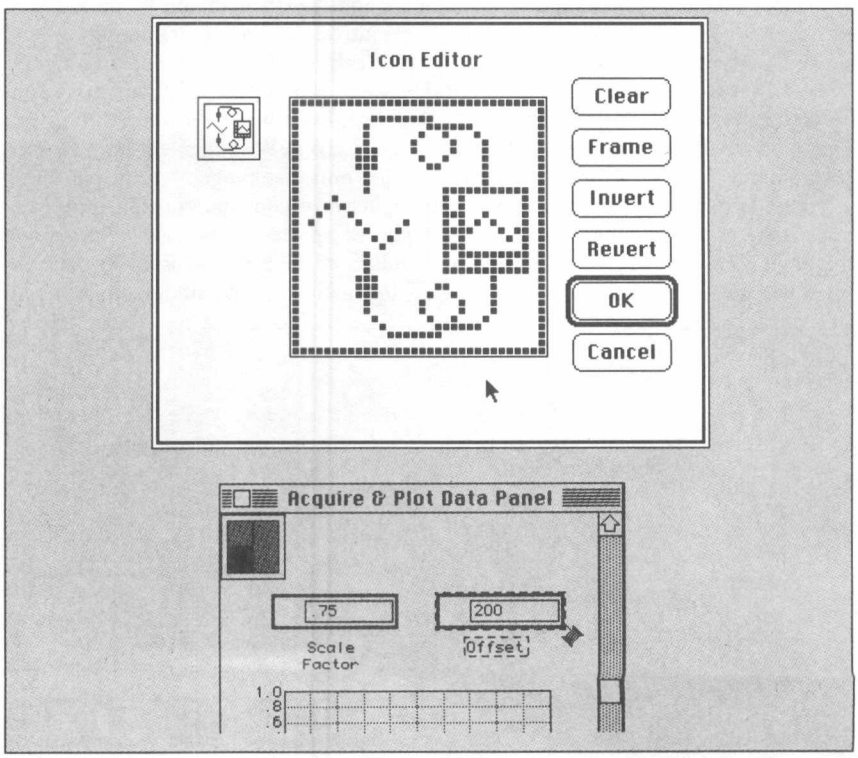


Figure 9: *Creating a programmatic interface for a virtual instrument so it can be used as a dataflow instruction in another virtual instrument. First, an icon is created for the virtual instrument using the built-in icon editor (top). Then its front-panel controls are associated with areas on the icon (bottom). Finally, the virtual instrument is saved to disk in its own file.*

Embedded Systems Design — A Special CASE?

Integrating tools for real-time systems development calls for more than just glue — it calls for a systems approach

by David Kalinsky and Jim Ready

Over the past five years, scores of useful tools have been introduced for the development of embedded systems software. Ranging from front-end CASE tools, through host- and target-based implementation tools, to performance analyzers and in-circuit emulators, most phases of software development have been addressed by tools vendors.

Although additional capabilities in each category will certainly advance the state of state-of-the-art in software development, what is sorely needed at this point is integration between this plethora of tools. Currently, developers must move from toolset to incompatible toolset as they progress through the life cycle (see Figure 1, page 38).

Integration of these toolsets with each other, and with knowledge of the run-time software and hardware environment, will vastly increase the practicality of developing large-scale embedded systems software. Today, vendors are working to integrate traditionally disparate toolsets (CASE tools, microprocessor development systems, compilers, and debuggers) with technologies to provide smooth interfaces between stages of development. Needs include robust sets of tools for each stage and for each transition between stages of the software development life cycle (see Figures 2 and 3, pages 39 and 42). Each must include the extra functionality required to deal with the complexities of embedded systems (such as object-oriented design, multitasking synchronization and communication, multiprocessor target systems, timing requirements and analysis, and deterministic response).

This article examines an embedded system software development cycle using typical current tools technology, and highlights where tools integration has historically been neglected. Technology trends that improve integration (thereby reducing software development and maintenance costs) are described.

David Kalinsky and Jim Ready can be reached at Ready Systems Corp., 407 Portrero Ave., Sunnyvale, CA 94086.

System Requirements and Design

Embedded systems development projects originate in a systems engineering phase. In this phase, software issues are not yet

addressed because the entire system is being thought about abstractly as a "black box." In an application area such as avionics, the "black box" may actually be envisioned as being painted orange or dark green and being connected by ruggedized connectors to other parts of an airplane. In medical instrumentation, another embedded application area, the "black box" may have an attractive man-machine interface at its front panel, and cables to sensors and other medical devices at its rear.

The systems engineer, in close consultation with application area experts, begins work by analyzing system-level requirements. The engineer may want to express the functional requirements by using a graphic modeling technique (such as structure charts or data-flow diagrams). The engineer associates performance requirements with certain functions or chains of functions. For example, a certain stimulus-response path may be assigned an upper limit of time in which it must respond to a trigger. In the system requirements phase, the systems engineer will do modeling and simulation calculations to evaluate the feasability of the requirements.

A second step in system-level activity is called *system design* or *allocation*. In this activity, the systems engineer assigns to different technologies the individual function in the system requirements specification. For example, a function may be assigned to implementation in analog electronics, another to digital electronics, and others to perhaps electro-optics, or even to mechanical or vacuum technologies. Yet other system-level functions may be assigned to implementation as software on digital computers or microprocessors.

In a cardiologic system, for example, heartbeat detection may be assigned to an electronic peak detector, ECG presentation to a digital-analog display subsystem and waveform analysis and alarm detection to software running on an embedded microprocessor.

Once an initial allocation has taken place, it must be verified that every system-level requirement has been allocated to one (or more) system components. It also must be verified that there are no superfluous system components that do not work to satisfy some system requirement. These checks are often termed *traceability*.

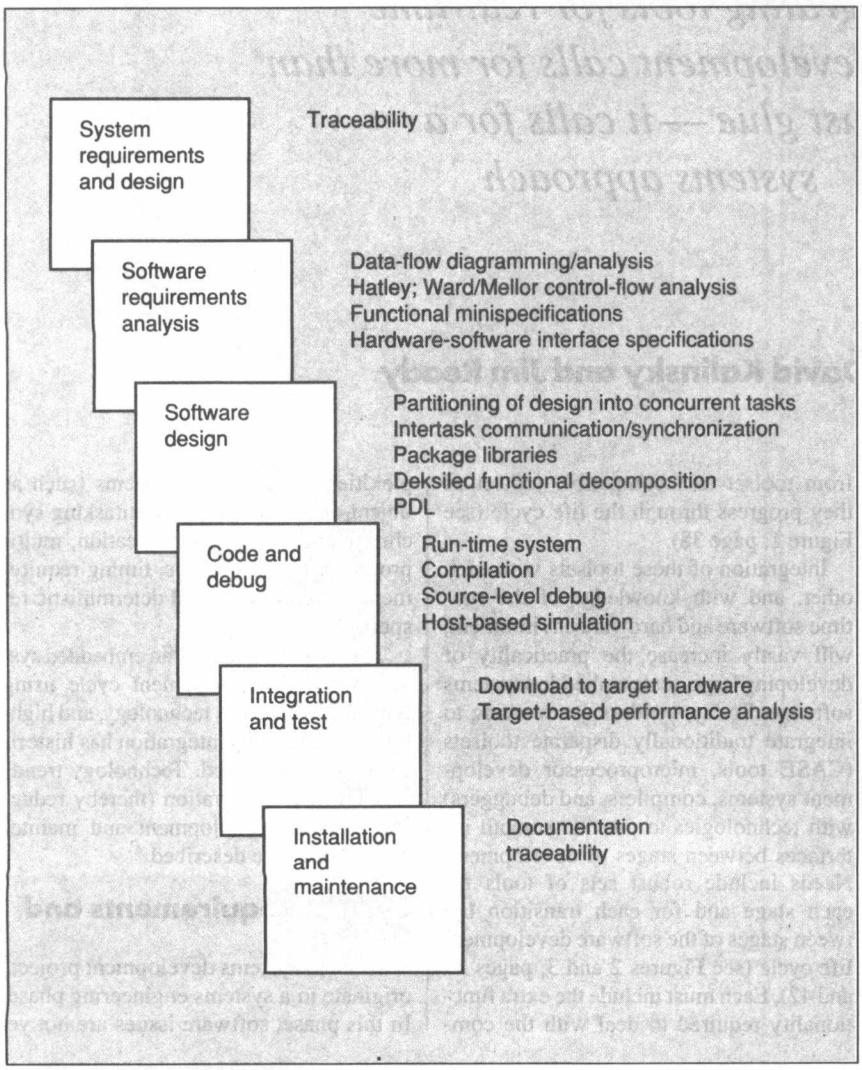


Figure 1: *Embedded system/software life cycle versus tools needs*

Software Requirements Analysis

Once system-level allocation has taken place and computer software has appeared in the system design, the work of the software engineer formally begins. The software engineer, in cooperation with systems engineers, must develop requirements for this software based on those system-level requirements allocated to it, and interface needs that have appeared in the system-level allocation.

The primary activity of the software engineer is to perform a functional analysis of the requirements of the software. This is usually graphically modeled using a data-flow diagraming technique in which functions are shown as bubbles, and their data connections are shown as arrows. Functional bubbles may be "exploded" to show greater levels of functional detail.

Together with the functional analysis, the software requirements engineer may need to describe the control interconnections between functions and perhaps model special "control centers" associated with the data-flow diagrams. Control-flow modeling techniques have recently been popularized by Paul Ward and Stephen Mellor and independently by Derek Hatley and Imtiaz Pirbhai. Both teams advocate the use of State Transition Diagrams and tables to detail the control logic.

CASE tool support is becoming available for these functional control modeling methods.

As part of the software-requirements

specification phase, all interfaces between computer software and its surrounding hardware must be fully specified. Such a specification becomes a detailed "contract" between the software developer on the one hand and the hardware developer on the other. Details down to the level of individual bits, interrupt addresses, and timing constraints must be specified. If these are not specified, the software to be developed will not integrate with the hardware to be developed in parallel.

The phase of high-level software design for real-time systems is an activity of major restructuring

The "interface control" specifications are being computerized at present. Typically, their computerization is not linked with the CASE-supported functional and control modeling methods. In the future, this link between tools will be vital in order to ensure that complex interfaces are specified consistently and completely and that they are kept stable and visible for all developers.

When the software requirements engineer finally presents the software requirements specification document, the systems engineer must be convinced that the software requirements specification addresses all issues allocated to it in the system-level allocation. The engineer must also be sure that the specification does not contain extraneous requirements that can not be traced back to system-level needs. Thus, traceability is again an issue at the software-requirements specification phase. Satisfactory tools for demonstrating this traceability have yet to emerge. When they do, these tools must be intimately linked to functional and control modeling tools. The components of functional and control models must be traced. Backward tracing is needed to link software requirements to system-level requirements and design models. Forward tracing is needed to link software requirements to design, implementation, and testing work further along the software life cycle.

Software Design

Once software requirements have been established, the software design process may begin. In embedded-systems real-time software development, the structure of the requirements model developed earlier is not the same as the structure of a design model that provides an implementation solution. Constraints such as timing requirements and special hardware-software interrelationships (such as interrupts and DMA interfaces) are typical reasons for the differences in structure of requirements and design models. The phase of high-level software design for real-time systems is an activity of major restructuring or even creating a new structure for the software solution.

This software-solution model typically includes the identification of concurrent tasks within the software and the selection of the appropriate mechanism for regulating each instance of intertask communication and synchronization. Data might be passed, for example, between tasks by way of mailbox, a queue, or an Ada rendezvous. Certainly data can not be passed freely between concurrent tasks because this would cause random-appearing errors of mutual exclusion violation. Control might be passed between tasks by using mechanisms such as semaphores, event flags, or an Ada rendezvous.

Today, most software designers mis-

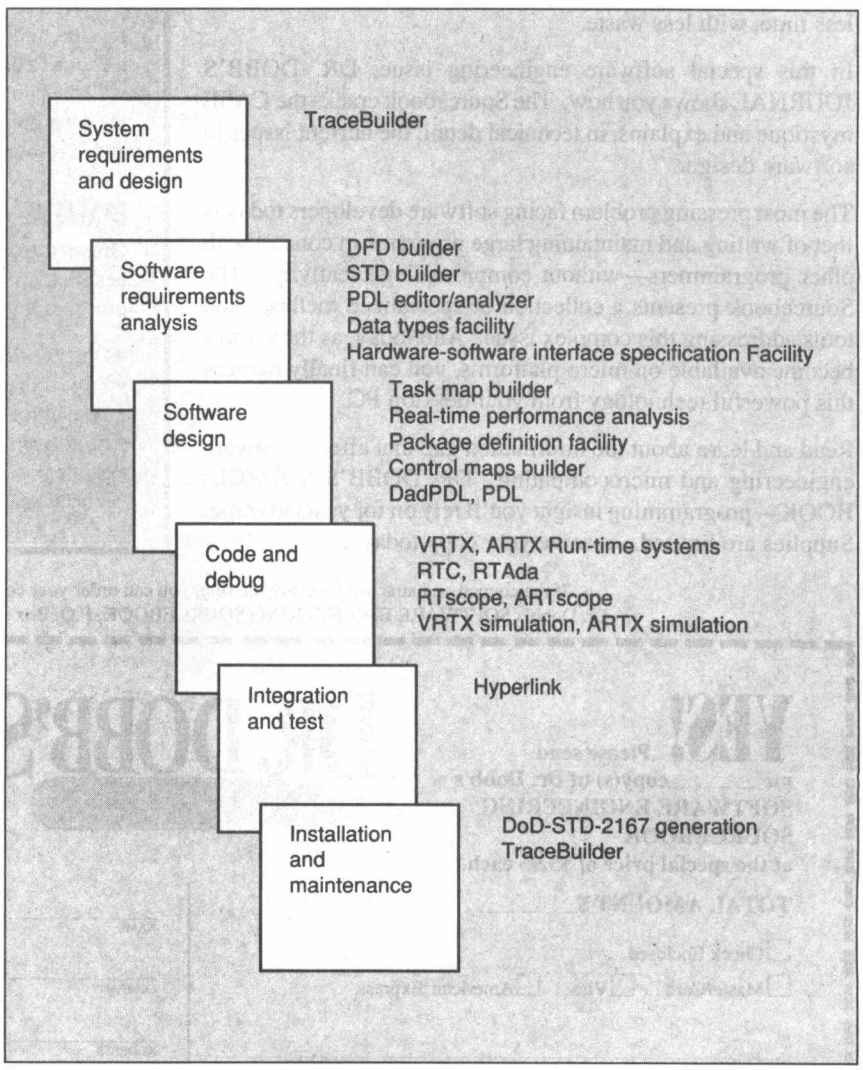


Figure 2: *Ready systems tools for each stage of the cycle*

(continued from page 39)
takenly use requirements modeling tools and techniques such as data and control flow diagrams for software design modeling. With such methods, much critical real-time design information (such as identification of tasks and identification of intertask communication and synchronization mechanisms) can only be noted as informal comments. Or worse yet, they remain as decisions stored only in the mind of the designer.

In the near future, the advent of real-time software design tools will occur. Such tools explicitly capture these sorts of real-time design decisions in the form of visual formalisms, which are both well-defined and easy to absorb because of their graphic expressions. Eventually these tools will also make it possible to perform timing analyses of these visual depictions of high-level real-time designs. The basis for these analyses will be intimate knowledge of the underlying executive or run-time software, as well as the hardware environment, built into the design/timing analysis tool. Early design tools will provide the ability to do static timing calculations on these diagrams in order to evaluate whether a software design will meet real-time constraints. More advanced design tools will eventually perform dynamic simulations, which may even be viewed as animations of the tasking diagrams.

Such tasking diagrams may be constructed and detailed by using either hierarchical or object-oriented methods. Tools today provide strong support for object-oriented methods. What the software designer really needs is a toolset that allows selection of a method as it is most appropriate for each section of the project, and even mix-and-match alternative methods.

Also important is the issue of continuity between the software design phases and other life cycle phases — in other words, traceability. Every entity in a design must be justified by a firm tie to a need (or needs) expressed in the requirements models. The entity must be developed into a specific piece of code that it specifies.

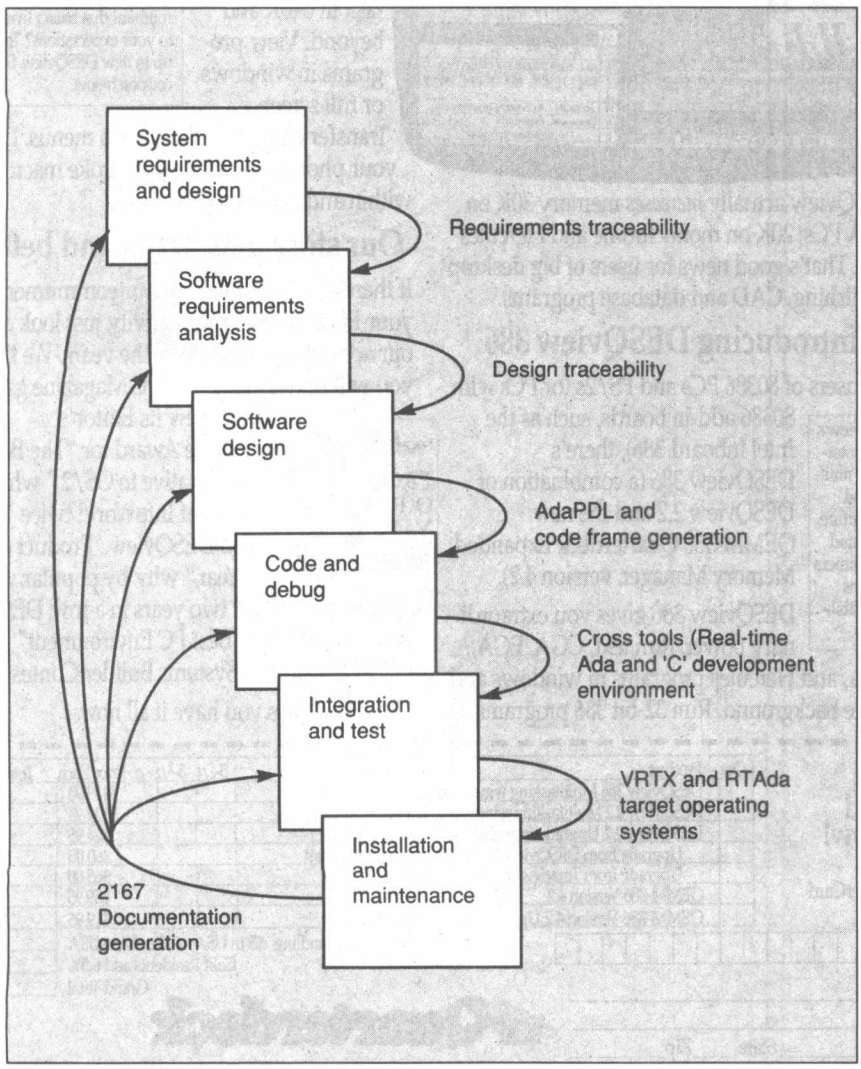


Figure 3: *Ready Systems tools link the life cycle*

Code and Debug

Today's implementers manually translate individual pieces of a software design into source code. This is not only tedious but inevitably leads to a rift between the software design and the software code implementation.

As software design tools progress toward higher levels of integration (for instance, between high-level structural models such as task diagrams and low-level detailed specifications such as program design language), it is becoming possible to envision automatic translation of design specifications into source code. In the sequential-programming world of data-processing applications, this dream is close to reality. In the highly constrained world of real-time software, this dream is only beginning to be partially realized in code frame generators.

Such code frame generators must not only translate designs into code, but they must remain tightly tied to software design tools in order to assure that changes made directly in source code do not violate design specifications.

Code frames are manually filled out to complete source code which can be compiled. Languages and their compilers are among the best understood technologies in the software life cycle. Even in this area, real-time applications pose special problems in terms of the interrelation of programs with executive or run-time services, so that program tasks can run concurrently and can communicate and synchronize with one another. For example, the Ada language has compilers implement the rendezvous mechanism in such a manner that it is sometimes impossible to rely on an Ada run-time system to provide rapid and deterministic real-time response. Special executive kernels or run-time systems are being used to solve this

problem. Their characteristics must be evaluated in the timing analyses provided by real-time design tools.

Once code has been compiled and linked, and its relations to an executive kernel or run-time system have been established, the software engineer would like to execute the code in order to begin debugging it. In embedded systems development, the target hardware is typically also early in its development and can not be used as a reliable platform for software execution. The software engineer demands simulation capability on the host computer on which software development is taking place.

Such simulation capability should not be restricted to the application code itself but must extend to a simulation of the underlying executive kernel or run-time system services in order to assess the real-time interrelations of concurrent tasks. For instance, a software engineer should at this stage reevaluate adherence to timing constraints, processor loading, and intertask communication and synchronization mechanisms usage.

The convenience of a source-level debugger is needed in order to make the software engineer's job easier at this stage. Such a debugger must not only be conversant in terms of application code, but it also must be "knowledgeable" of the executive kernel run-time system mechanisms (and their status) at all times.

As tools evolve and integrate, software engineers in the future will debug software directly from their high-level design graphic representations, coming in and out of levels of design detail and source-level implementation detail.

Integration and Test

Hardware development on the embedded system project eventually will progress to the point where it is feasible to attempt to integrate the software with the newborn hardware. This demands a high-speed link for efficient downloading of the software from the host to the target software. Efficiency in this connection is critical since many downloads will be needed as integration tests proceed and problems must be corrected. A slow downloading mechanism would force the software engineer into making changes by direct patching into the executable code on the target hardware. This would create an irreparable rift between target code and source code, and contradictions with design and perhaps requirements.

To combat these dangers, the high-speed link should also be used as the access channel for a host-based source-level debugger of target-based code, which would complement any target-based debugger. This host-based debugger would tie the executable code back to design and requirements specifications so that the code could be viewed from these various high-level perspectives and kept consistent with them. For example, a target-based timing performance analysis could be tied back to design-based analyses and requirements-level constraints. A change in code should be tied back to the appropriate design component and to the corresponding requirements. In this way, the impact of any proposed change can be assessed, and an implemental change can be properly documented back through all the life cycle stages it affects.

Every entity in a design must be justified by a firm tie to a need (or needs) expressed in the requirements models.

Installation and Maintenance

As the development life cycle comes to its conclusion, system testing is performed to evaluate the degree to which a system meets its requirements. In order that these tests be as formal and complete as possible, they must be tied directly back to requirements. In other words, formal test plans and test results must trace to requirements. Few tools are available today to manage this traceability connection, and fewer yet can tie requirements-test traceability to requirements-design-code interrelationships. The ability to relate high-level test results to design and code components (by way of their tie-ins to requirements) would be a powerful technique for localizing errors and needed corrections.

This same connection, used in the same way, would be equally powerful for reducing the effort and cost involved in software maintenance.

Formal, consistent, up-to-date documentation, automatically generated from requirements and design and code information, is another leg upon which an efficient maintenance phase must stand.

Summary and Conclusion

Today's real-time embedded systems software engineer is faced with a project life cycle strewn with disparate, noninterconnecting tools. Starting with system and software requirements CASE tools, the developer must transfer the model to a toolset (or toolsets) more suited for software design. After that, the developer must manually translate the design into source code for a compiler. After compilation, the developer must find host-based debugger and simulation tools compatible with the object code generated by a particular compiler. Next, the object code must be quickly downloaded (by using a network) onto the target system and analyzed there for performance, efficiency, and bugs. Suitable, compatible download and analysis tools must be selected.

Today, coding is only a small fraction of the software engineer's effort. The engineer's main activities are modeling, testing, and documentation throughout the life cycle. While doing all of these activities, a significant proportion of the engineer's time is spent bridging rifts between the various tools that support these various activities. These bridges are jerry-built, hand-made improvisations that are totally unproductive efforts.

As methods and tools for more and more steps in the life cycle are better understood, an integration process is now taking place in the software engineering tools and industry. This integration will soon replace those unproductive bridges and provide the real-time embedded systems software engineer with a seamless set of software development tools.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Shift
Lock
End
Ins
Del
PgUp
PgDn
and
relation
relation
relation
relation
PUNCHED TAPE
MANUAL INPUT
TERMINAL
DISPLAY
CONNECTOR
DOCUMENT
DECISION
DRUM
PROCESS
INPUT/OUTPUT
ONLINE STORAGE
PREPARATION
MANUAL OPERATION
TENTHS
CARD SCALE 100

Tackling Large-Scale Programming Projects

Exploring the power a networked distributed computing environment brings to the software development project

by William Courington, Jonathan Feiber, and Masahiro Honda

When the personal computer revolution started back in the 1970s, life was relatively simple. The computers, though more difficult to use than many of today's PCs, were relatively simple machines. The software these computers ran was also simple, largely because of the computers' limited memory and other resources. This software was usually developed by one person.

As the complexity and power of personal computers grew and their use became more widespread, users demanded software that was more sophisticated and easier to use. Nowadays, developing new software products usually involves large teams that include not only programmers, but quality assurance engineers, release engineers, technical writers, and others, all working toward a common goal.

These large-scale programming efforts, often undertaken in a network environment, face several challenges not encountered by the lone developer. Developers now must contend with the difficulties of product complexity, staff interference, and network utilization.

Product complexity involves not only the profusion of files associated with a product, but also the management of different versions of these files and the ability to rebuild these different versions. Staff interference comes about from the interaction of a project team, such as when two or more programmers want to modify the same module at the same time. The added dimension of working on a network brings its own problems, as well as benefits.

Over the years, several tools have been developed to address these problems. The most promising solution today is an object-oriented, network-based, interactive environment, such as the one present in the Network Software Environment (NSE) developed by Sun Microsystems. Using such a system, a complex product appears to be composed of a few components; people modifying the same component appear to have their own copies; and resources scattered around the network appear to be local to each machine.

Objects Make Complexity Manageable

One characteristic of a large-scale software product is a profusion of files. A single project may have hundreds or thousands of source files, object files, libraries, executables, and documents. To impose some order on this mass, developers typically put related files into directories and organize the directories into hierarchies reflecting the product's structure. For example, the files related to a subsystem might be placed in a common directory, and files related to programs in that subsystem might be placed in subordinate directories.

Although a hierarchy of directories is a well-proven organizational tool, it does not address several other product-complexity issues.

As a product evolves, so do its files. Each file typically exists in multiple versions, with each version representing a set of enhancements or bug fixes.

Managing multiple versions of files is the domain of tools known as version control systems. The original and perhaps best-known of these is SCCS, the Unix system source-code control system. Like directories, version control systems help manage one of the many dimensions of product complexity.

If multiple versions of multiple files are not trouble enough, the time required to rebuild a large system (compile and link all modules) is often so great that complete rebuilds are impractical on a routine basis. The alternative to rebuilding an entire system is to find the source files that have been changed, recompile them, and then relink the system. Although this is simple in principle, faithfully tracking changes is difficult in practice, and imperfect tracking can produce bugs of exquisite obscurity when incompatible modules are linked together.

A class of tools sometimes called system modelers has been developed to automatically find and rebuild only changed files. The Unix *make* program is the best-known system modeler, and it, too, addresses one more aspect of product complexity.

Directories, version control, and system modeling represent the state of the art in many software projects today. Disciplined use of these tools is a great help, but it is still inadequate because the tools are exclusively file oriented. When the complete software development cycle is

W. Courington is a technical writer for Sun, holds an A.B. from Occidental College and has been writing about software for 15 years. J. Feiber is the director of programming technologies for Sun and holds a B.A. in computer science from the University of Colorado and has eight years experience in software engineering. M. Honda is manager of NSE development at Sun. He has a Ph.D. in computer science from the University of Wisconsin at Madison and has 12 years of experience in software engineering.

considered, it becomes clear that a software product consists of more than just code-related files. The product may also include proposals, schedules, requirements, drawings, documentation, specifications, data dictionaries, test data, test drivers, and test results. There needs to be a way to manage these diverse entities, which may be manipulated with tools supplied by multiple vendors, in a coherent way.

A single general solution to accommodate this diversity of software building blocks can be found in a hierarchy of objects. Everything in an object-oriented system is an object of some type. All objects have names, values (contents), and revisions (change histories).

Figure 1, below, shows a typical object hierarchy. Three types of objects have been defined: files, targets, and components. Additional object types — for example, data flow diagrams or data dictionaries — can also be integrated into such a system. All object types, whether developed in house or by outside suppliers, fit into the object hierarchy. Note especially that components can contain all types of objects, including other components.

Files, Targets, and Components

At the bottom of the hierarchy are files. Sun's NSE distinguishes between three types of files: ordinary, source, and derived. Ordinary files are just that; no special facilities are provided for them. Source files have their version history maintained, allowing any version of a source file to be recreated at any time. Summaries of the differences between two versions or of the changes made to successive versions of a source file may be displayed. To minimize disk space, the NSE stores only the differences between successive versions of a source file.

Derived files are created by programs, usually compilers or linkers. Object files, libraries, and executables are examples of derived files. The NSE does not maintain successive versions of derived files because any version of a derived file can be recreated from the corresponding source file version.

Targets encapsulate all the files necessary to build a derived file together with a recipe for building it. The membership of a target must be kept up to date. For example, if a source file is changed to include a new header (include) file, the header must be added to the target. The NSE handles this automatically. Target derived files are built under the control of the *make* program described earlier. (The recipe is actually a makefile.)

Components are the most versatile objects because they can contain other objects, including other components. This property is similar to directories containing files and other directories. Because components can have other components as members, they are the natural object for representing the structure of a software product. For example, one group of components might represent a system's major subsystems. These components, in turn, might contain components representing minor subsystems. These last components might contain still other components representing programs.

The most basic component might contain only a target. A more elaborate component could also contain objects related to the target, such as a test-data file, an executable test driver, or a documentation file. In short, any related objects that are naturally examined and changed as a unit are good candidates for grouping together in a component. (The issue of which objects belong in a particular component should be decided by local preference, the same as when determining what goes into a directory.) Components can freely share members, allowing, for example, one copy of a common header file to be shared across many components. Revisions of components are maintained in a manner similar to that used for source files. Old revisions are immutable and can be accessed at any time. When a component is revised, all subcomponents are automatically revised too.

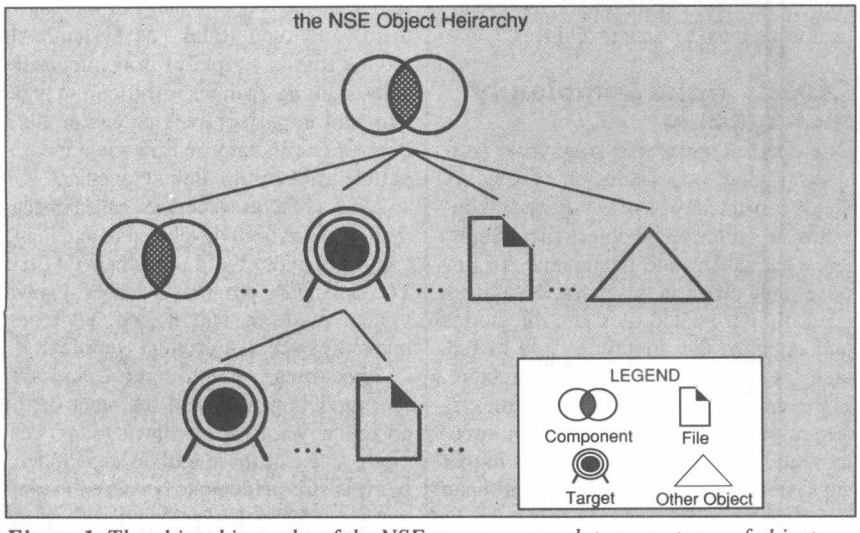


Figure 1: *The object hierarchy of the NSE can accommodate many types of objects*

Environments Insulate Activities

A component hierarchy can make the structure of a complex software product intellectually manageable, but it does not address the problems that arise from the interaction of project team members. For example, if two programmers share a copy of a source file (or any object), changes made by one programmer are, at the very least, destabilizing to the other's work. The alternative to sharing is copying, which has its own interaction problems. If, for example, two programmers modify local copies of a master file and replace the master with them, the second programmer's changes obliterate those made by the first.

NSE has a facility, that is called an environment, for controlling concurrent access. Unlike traditional concurrency control mechanisms, such as locks, environments are "optimistic" and don't assume that the changes are incompatible. Suppose two programmers need to change the same source file at the same time: One programmer needs to fix a bug while the other wants to add a procedure. A "pessimistic" concurrency control mechanism, in effect, assumes that the changes are incompatible and allows only one programmer to change the file at a time. Environments, on the other hand, allow both programmers to change the same file

in parallel so long as they make the changes in different environments. Eventually, of course, the two sets of changes must be merged into a new version of the file. The NSE provides a tool (described later) that automatically merges compatible changes and provides assistance for resolving incompatible changes. Thus, different environments enable programmers to work on the same file in parallel, and the time spent merging the changes will be proportional to the degree of incompatibility of their changes. Generally, as a software product matures, incompatible changes become increasingly rare, requiring very little time to merge changes made in parallel.

Virtualizing the File System

An environment is an insulated programming workspace that is, in many ways, similar to a process' virtual-address space in a virtual-memory system. Within an environment, a programmer refers to files by their ordinary names and manipulates files with ordinary tools such as compilers, editors, and utilities. However, just as the memory mapping hardware of a virtual-memory computer transparently maps the same address reference made by two different processes into different *physical* addresses, so the NSE transparently maps references to the same file from different environments to different underlying files. The result: A change made to a file in one environment is invisible in other environments, even though they contain the same file.

Environment Hierarchies

Environments can be arranged in hierarchies, as shown in Figure 2, page 49. An environment hierarchy reflects the relationships among project activities. That is, all environments represent activities, and subordinate environments reflect subacitivites that can proceed in parallel with their parent activities. Thus, a diagram of a project's environment hierarchy will strongly resemble the project's organization chart. Components are distributed through an environment hierarchy as they are pertinent to the activities conducted in each environment.

Although all environments have the same properties (there are no environment types), they can be grouped into three classes corresponding to activities common to many software projects: re-

(continued from page 47)
lease environments, integration environments, and development environments.

At the top of an environment hierarchy is a release environment. It is here that the complete product is tested and released to manufacturing or to customers. A release environment contains the component(s) at the top of a product's object hierarchy; these components contain all other components as subcomponents, but the subcomponents are not the concern of the person or department in charge of the release environment.

To begin development of a new major release, a release engineer can create a new release environment as a child of the current one. Copying the latest revision of the components from the old release environment to the new one establishes a new baseline for development.

Integration environments typically correspond to the activities of project groups such as departments. An integration environment usually contains the components (and their subcomponents) comprising a subsystem. A project that has multiple group levels can create corresponding levels of integration environments.

At the lowest level of an environment hierarchy are development environments, which have no children. These are the workspaces of individual staff members, and usually contain the components they are actively working on. The basic business of a software project, changing and compiling source files, is conducted in development environments.

Detecting and Resolving Conflicts

As mentioned earlier, some mechanism must exist to reconcile conflicting concurrent revisions. The NSE provides two operations, called acquire and reconcile, that logically copy components (including their subcomponents) from parent environments to children, and vice versa. (The term logically copy emphasizes that although the user of an environment sees what appears to be local copies of files, to the degree possible, the NSE shares the same copy of files among multiple environments.) The acquire operation copies components from a parent environment; the reconcile operation adds a new revision of a component to the parent environment.

Thus, to modify the current revision of a component, a programmer acquires the revision from his or her department's integration environment. When the modifications are complete, the programmer reconciles the component back to the parent integration environment. Reconciling the components in the top-level integration environments into the release environment produces a new revision of the complete product. Development can continue in the lower-level environments while the new release is being prepared in the release environment.

The NSE encourages parallel development, and does not prevent two programmers from acquiring the same component from a common parent environment, modifying it, and reconciling it back. If, for example, two programmers both acquire revision 5 of a component, each with the intent of creating revision 6, whoever reconciles first does create revision 6 in the parent environment. When the other programmer reconciles, however, the NSE detects a potential conflict. Revision 6 must be merged with the second programmer's changes; the merged result can then be reconciled to the parent to create revision 7. The result is exactly the same as if the programmers had created revisions 6 and 7 sequentially — except that by working in parallel, they will probably finish the job sooner.

The NSE resynch operation effectively acquires the conflicting revision of a component so the conflict can be resolved in the child environment. With conflicting revisions in the same environment, the file resolve tool automatically merges nonconflicting lines from two source file versions and marks conflicting lines for manual resolution. When all conflicting files are resolved, the component can be compiled, tested, and then reconciled back to the parent environment.

Object and environment hierarchies effectively address product and project problems on a single machine, but a network adds yet another dimension to these problems. The new dimension is exemplified by this question: Where in the net is environment ABC, and where is revision 3 of component PDQ? The NSE addresses this question with a network-wide naming scheme. Users refer to environments and objects by simple names; the NSE transparently maps these local names to network addresses and, using Sun's Network File System, automatically mounts and accesses remote file systems as needed.

Summary

The NSE addresses many of the difficult problems that arise in large and complex software development projects. The NSE manages the complexity that comes from a large number of files, multiple versions of each file, multiple revisions of a product in the field, and multiple processor architectures. The NSE does all of these things in a networked, distributed development environment. Although not described in this brief article, the NSE supports the integration of tools that cover every phase of the software development lifecycle. Perhaps most importantly, the NSE understands the issue of scale; it continues to support projects as they grow from tens to hundreds of programmers, and from thousands to millions of lines of code.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

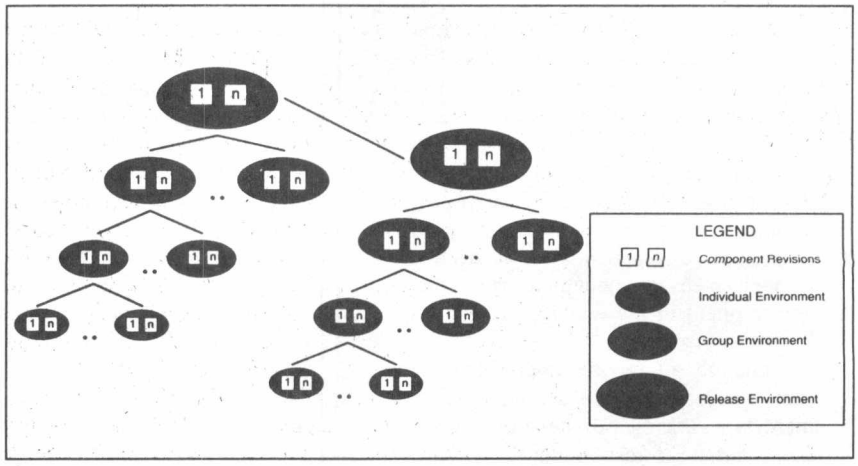
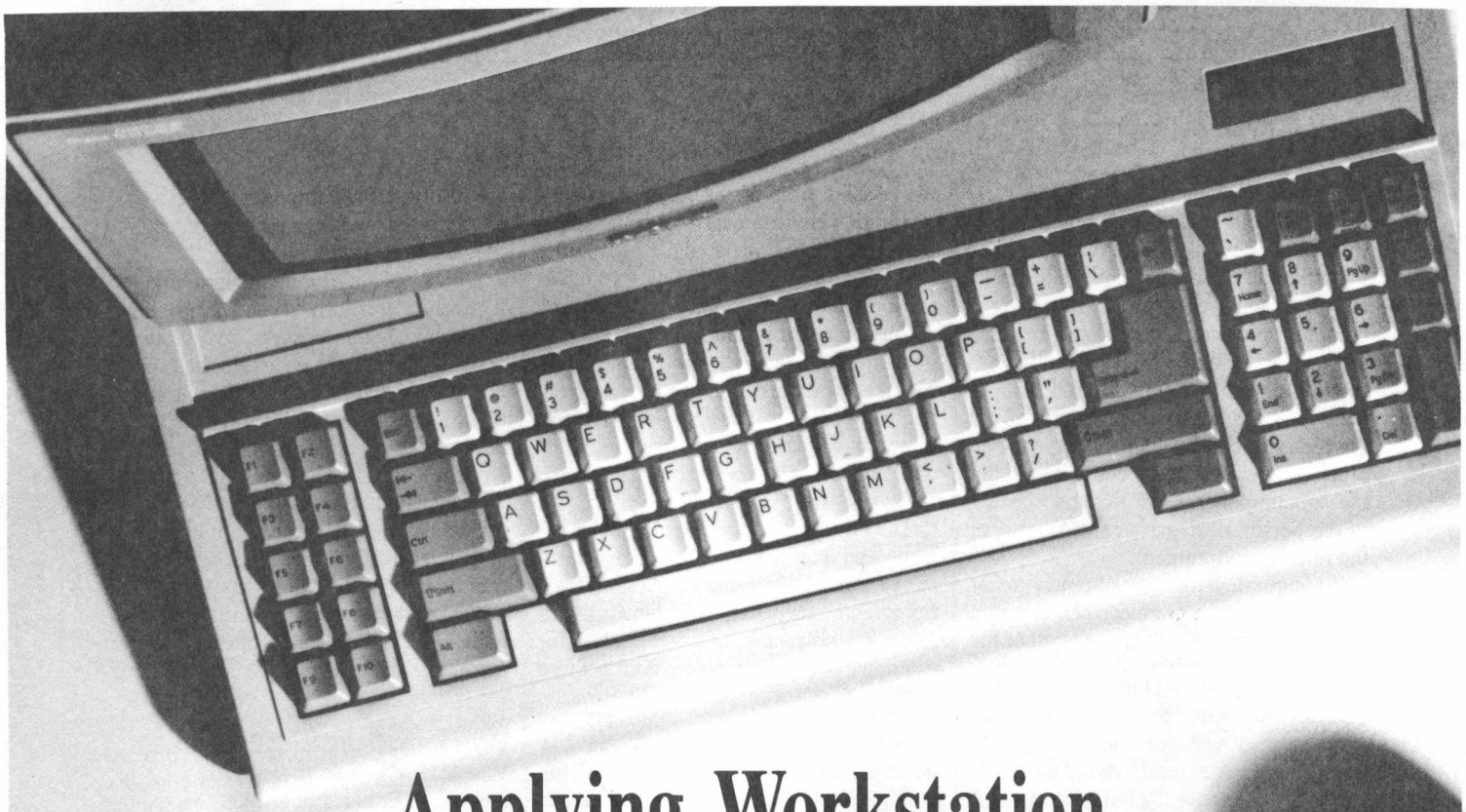


Figure 2: *An environment hierarchy mirrors the relationship among project activities*



Applying Workstation Technology to CASE

Large-scale programming power through a distributed development environment

by David B. Leblang

To manage the complexity of a large software development effort, a project team requires many specialized tools in order to finish on time and within budget. A project would likely use requirements analysis and design tools, document editors, coding and debugging tools, project management, and cost estimation tools, as well as a configuration management system. These tools must work together to form an *integrated project support environment*. In addition, since large projects are distributed over many computer systems these tools must also recognize and support a distributed development environment.

Networked workstations, with distributed operating environments, provide the computing power and graphics power necessary for sophisticated CASE tools while also providing the high degree of data sharing needed by a large development team. By combining the techniques of *object-oriented programming* and *distributed systems programming* a new class of high productivity CASE solutions can be built that take full advantage of the graphics, network, and computation power of modern workstations.

This article describes some general problems encountered in large development projects and some ways the power of distributed computing and the integration of CASE tools can be used to overcome them.

David B. Leblang is a senior engineer at Apollo Computer Inc. He has been with Apollo since 1982 where his interests include software configuration management, computer-aided software engineering, and distributed computing systems. He received a M.S. in computer science from Boston University, and an S.B. in computer science from M.I.T. Formerly, he worked at DEC Research. He is a member of the IEEE Computer Society and a member of the ACM.

Large-Scale Problems

Large-scale development has particular characteristics not usually found in smaller efforts. In large efforts the system architects and designers are often outnumbered by programmers, technical writers, and sometimes even by marketing personnel. Although the designers may also write code, outline documentation, and help with market brochures, the bulk of the work will be done by people who do not have the designers' intimate insight into the operation of the software. In fact, with very large applications there may be no single person who understands the entire system.

Under these conditions the only way to succeed is to write down the requirements, functional specifications, and design in such a clear and unambiguous

way that many small teams can proceed independently and still have their pieces fit together later. The high level of common understanding achieved through a common set of requirement and design documents makes the coordination of individual development teams possible.

Large-scale efforts also require version control and configuration management. During the development of a large project, the requirements, design, code, and other objects change as the project progresses. For example, in a recent project using Apollo's DSEE configuration management system, there were 3,600 changes made in 18 months by 20 different people to a relatively small source code library (100,000 lines); the average code module now has 40 versions. The source library was only one of approximately 50 libraries used to build the product. Some libraries had more than ten variations of the code being worked on at the same time, and one particular library, shared by many projects, had 330 different people access it in the last year. Obviously, a powerful version control and configuration management system is needed.

Large-Scale Solutions

CASE tools evolved from the need to solve the unique problems of large-scale development efforts. These tools generally fall into one of three categories: development tools, process tools, and integration tools.

CASE development tools are oriented toward particular phases in the software life cycle (see U.S. Department of Defense, 1985), including the definition of requirements, design analysis, coding, testing, debugging, and more. CASE process tools are oriented toward activities that are common to all life cycle phases (for example, version control, configuration management, and project management).

The distinction between development tools and process tools is not always clear. CASE integration tools maintain abstract relationships between objects managed by individual CASE tools. (For example, relating a requirement specification to the design modules that satisfy the requirements, or relating a design module to the source code modules that implements the design.) Integration tools help to bridge the gap between life cycle phases and to support the tracing of requirements and project management.

Mechanisms that can relate the CASE objects managed by different tools is a current hot topic in the CASE community. A discussion of the issues and some possible solutions are presented later in this article.

Workstation CASE Tools Versus PC CASE Tools

Some CASE tools run on PCs (for example, requirements and design editors), but these tools tend to be oriented toward

WORKSTATION TECHNOLOGY

single users. They do not address the issues of large-scale distributed development. For example, workstation CASE tools tend to address the problems of version control, distributed and concurrent access to objects, and the protection of objects. Large projects will define thousands of design and code objects, as well as other objects, which will be distributed across many machines. Unlike PC CASE tools, workstation CASE tools must operate in a distributed environment in order to manage their objects and the relationships of those objects.

The software architecture of these tools is usually quite different from that of a single-user tool. The workstation tools often run as multiple processes; one process may act as a server that provides access to a database while many users run client processes that talk to the database server. Another architectural difference is that the tools are written to run in a very large virtual address, rather than being squeezed into a small physical address space. The larger displays available on workstations are used to provide more sophisticated user interfaces.

Although PCs are evolving toward larger address spaces, larger displays, and better support for networking, the basic single-user philosophy still persists. In some sense a workstation is a very high-end PC with a network connection. The CASE applications that take advantage of the high-end features will be more useful in a large project environment.

A "CASE workstation" today usually runs Unix and is configured with a 1- to 7-MIP CPU, around 8 Mbytes of memory, about 200 Mbytes of hard-disk storage, and a 19-inch, 1280-by-1024 bitmap display with multiple windows. These systems are always connected to a 10- to 12M-bit network, and they generally cost between $5K and $10K. They are approximately five to ten times as powerful as an entire timesharing system from the late 1970s. By the early 1990s, workstations will have performance exceeding 100 MIPs and a network bandwidth of over 100M-bits.

Workstations are used in large development efforts such as aerospace systems, telephone switching systems, automotive electronics, and other projects involving hundreds of people and millions of lines of code. They are also used by much smaller projects that require good team coordination.

The challenge for the builders of workstation CASE products is that they must find a way to exploit the hardware advances that are producing faster CPUs, faster networks, and larger displays. They must build software products that increase individual and group productivity.

Network Computing Technology

A large development team, perhaps as large as 300 to 500 people building several million lines of code, consists of many members that need to share code, design, documents, and other types of

objects. A transparent distributed file system enables a user to access a file stored on any machine in the network as if it were a local file. A workstation environment with transparent distributed file access (see Leach et al.,1983) allows team members to have their own dedicated computers and still share a single (very large) file system. This combines the best of time-sharing and personal computing. It increases productivity by eliminating the error-prone process of maintaining multiple copies of files (and of trying to keep them synchronized by exchanging floppies). Any project that is large enough to have sources and documents on multiple disks will benefit from a transparent distributed file system.

A network computing environment allows users to share computing power as easily as a distributed file system allows them to share files. A network computing environment also makes it easy to write client/server-based tools. This is important for CASE tools supporting large development efforts because the tools need to coordinate their access to such shared resources as a database. Apollo's Network Computing System (NCS) (see Dineen et al., 1987) is a portable implementation of a network computing environment that runs on Unix, MS-DOS, VAX/VMS, and other systems. It enables different types of machines, from a 16-bit IBM PC to a 64-bit Cray supercomputer, to share computations as well as data. This allows a network of computers to function as a single large computer by providing heterogeneous distributed computing services (see Notkin et al.,1987).

Key facilities provided by NCS include remote procedure call, service-location brokerage, and data conversion. Remote procedure call (RPC) is a mechanism that lets an application make an ordinary looking procedure call that executes on a remote machine. Expensive operations like image rotation and database queries can be executed on special-purpose servers, and in addition, several operations can be executed in parallel on different machines. Service-location brokerage acts like a "yellow pages," enabling a client program to find a service on the network. Automatic data conversion smooths the differences between data representations on the various machines, making it easier to exchange parameters.

NCS also provides an interface definition language and stub compiler. These help the user define the client/server interface and generate the code that converts ordinary procedure calls into message-passing operations.

Integration of CASE Tools

A large project will use many different CASE tools, each with its own user interface, database of objects, and function. Users want all of the tools they use to work together. The vagueness of this statement reflects the fact that there are actually several different problems that need solving.

"CASE integration" is a general term that refers to a set of solutions for these problems. There is no widely accepted CASE integration solution at present; however, there are many proposed solutions and partial solutions. The goal of a CASE integration facility is to provide a substrate for the CASE tools of many vendors.

One common problem is the portability of CASE tools: a project wants to use several different CASE tools but can't unless they all run on a common platform. The widespread use of Unix for workstations is helping with the portability problem. In addition, a variety of portable CASE tool interfaces are being proposed; these would be layered on top of existing operating systems. Unix itself is not quite standard, so there are several efforts underway to standardize a version of it.

Since most CASE tools use graphics, a portable graphics interface is also needed. The X-Windows System graphics standard has rapidly gained acceptance. X-Windows does not, however, deal with the problem of variations in user interfaces, which can cause learning-curve discrepancies among users. A variety of "look and feel" standards are emerging on top of graphics standards to address this problem. These standards attempt to define the way menus and the mouse operate in general.

The ability to exchange bulk data between CASE tools, as CAD design tools exchange data with CAD board layout tools, has led to the acceptance of the EDIF standard for information interchange. Extensions to EDIF and other data exchange standards are under discussion.

Many workstation CASE tools have adopted an object-oriented user interface; that is, they represent their objects (project tasks, design modules, documents, and so on) as graphics symbols on the screen. Users interact with the tool by pointing at objects and clicking mouse buttons. Design and project management tools tend to be bubble- and arc-oriented. The bubbles on the screen represent objects, and arcs represent relationships between the objects. For example, if one task must be completed before another task can be started, a project management system may indicate this by drawing an arc between a bubble that represents the first task and another bubble that represents the second. Users can navigate among the bubbles, create new bubbles, and perform various types of analysis on the bubbles and arcs.

Other tools, like documentation and code management tools, have desktops with icons that represent objects. Still other tools may have a different way of representing the objects it manages. Although the details of user interfaces may vary, each tool displays its objects and the intra-tool relationships between them.

Perhaps the most fundamental problem that needs to be addressed by CASE inte-

gration is the object correspondence problem. "Object correspondence" refers to the problem of relating objects managed by one tool to objects managed by another (for example, relating requirement paragraphs to design specifications and the code module that implement it). This is a major problem for manual systems and a primary job for CASE integration systems. Although intra-tool relationships are maintained by individual CASE tools, inter-tool relationships must be maintained by a third-party agent. A standard for the CASE tool/third-party agent interface is currently being discussed in many forums. Ultimately, users want to be able to create and navigate inter-tool relationships as easily as they create and navigate intra-tool relationships.

A large project may contain thousands of objects with tens of thousands of relationships. As new objects and new versions of objects are created, the relationships must be maintained. Requirements often evolve as a large system is built; verifying that the design and code match the requirements is known as the "requirements traceability problem" and is a special case of the object correspondence problem.

Intra-tool Versus Inter-Tool Relationships

Large projects use many tools, which are often supplied by different vendors. Maintaining the relationships between the objects managed by these tools requires some degree of standardization. A variety of integration schemes are currently being debated. In addition to solving requirements traceability problems, it would be useful to let a user point at one object on the screen and have the integration facility navigate to related objects. This is very similar to HyperText or HyperCard.

Conclusion

Large development efforts require sophisticated tools in order to succeed. Networked workstations provide the opportunity for a class of tools, known as distributed CASE tools, that address the problems of large-scale efforts. These tools tend to use a large graphics display to provide a good user interface. They operate in a network environment, support version control and concurrent access, and tend provide the "hooks" needed to support CASE integration tools.

Emerging standards in the areas of graphics and user-interface styles, portable operating system interfaces, and network computing environments are helping to bring CASE tools closer to the ideal goal of an integrated project support environment (IPSE). The 1990s will see continued growth in the availability of workstation-based CASE tools.

References

Brooks, F. *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1975.

DeRemer, F. and Kron, H. "Programming in the Large Versus Programming in the Small." *IEEE Transactions on Software Engineering*, vol. 2, no. 2, (June 1976), pp. 80 – 86.

Dineen, T.; Leach, P.; Mishkin, N.; Pato, J.; and Wyant, G. "The Network Computing Architecture and System: An Environment for Developing Distributed Applications." *Proceedings of the Summer 87 USENIX Conference*, June 1987.

Leach, P.; Levine, P.; Dorous, B.; Hamilton, J.; Nelson, D.; and Stumpf, B. "The Architecture of an Integrated Local Network." *IEEE Journal on Selected Areas in Communications*, November 1983, pp. 842 – 857.

Leblang, D.B. and Chase, R.P., Jr. "Parallel Software Configuration Management in a Network Computing Environment."*IEEE Software*, November 1987.

Leblang, D.B.; Chase, R.P., Jr.; and McLean, G.D., Jr. "The DOMAIN Software Engineering Environment for Large-Scale Software Development Efforts." *Proceedings of the 1st International Conference on Computer Workstations*, San Jose, Calif.: IEEE Computer Society, November 1985, pp. 266 – 280.

Liskov, B. and Zilles, S. "Programming with Abstract Datatypes." *ACM SIGPLAN Notices*, vol. 9, no. 4 (1974).

Notkin, D.; Hutchinson, N.; Sanislo, J.; and Schwartz, M. "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Hetrogeneity." *Communications of the ACM*, February 1987, pp. 132 – 140.

U.S. Department of Defense, DOD-STD-2167. *Defense System Software Development*. Washington, DC 20301: June 1985.

Wirth, N. "Program Development by Stepwise Refinement." *CACM*, vol. 14, no. 4 (April 1974).

Yourdon, E.L. *Constantine Structured Design*. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

A

ACT and Battlemap
Type: Maintenance
Performs complexity analysis and maintainability measurement
User: Analysts
McCabe & Associates
Twin Knolls Prof. Park
5501 Twin Knolls Rd., #111
Columbia, MD 21045
800-638-6316
PC compatible
$10,000 base price, call for details

APS/PC Workstation
Type: Prototyping, development, maintenance
Allows user to generate applications for automating systems development
User: Analysts, programmers
Sage Software Inc.
3200 Monroe St.
Rockville, MD 20852
301-230-3200
PC, XT, AT, PS/2, and compatibles
$4,500 to $6,000, depending on options selected
Notes: A full-function PC workstation for developing Cobol-based applications,

APS/PC Workstation allows the developer to design, prototype, generate, and unit test Cobol applications on a PC.

AdaGen
Type: Prototyping, development
Automates and integrates Ada systems
User: Developers, designers
Mark V Systems
16400 Ventura Blvd., #303
Encino, CA 91436
818-995-7671
PC compatible
$2,500
Notes: AdaGen integrates analysis, high-level, and detailed-design phases, and is dBase III compatible.

AdaGraph
Type: Development
Develops complex Ada systems
User: Developers, systems analysts
TASC — The Analytic Sciences Corp.
1700 N Moore St., Ste. 1220
Arlington, VA 22209
703-558-7400
PC compatible
$9,875

Aion Development System
Type: Prototyping, development, maintenance
Develops applications in an expert systems shell environment
User: Developers, programmers
Aion Corp.
101 University Ave.
Palo Alto, CA 94301
415-328-9595
PC, XT, AT, MS-DOS
$7,000
Notes: The program is an expert system shell in which the user builds a knowledge base that is processed by an inference engine.

Analyst/Designer Toolkit
Type: Prototyping, development, maintenance
Supports Yourdon analysis and design technique
User: Designers, systems analysts
Yourdon Inc.
1501 Broadway, Ste. 601
New York, NY 10036
212-391-2828
PC, XT, AT, PS/2
$1,995

Analyst/RT
Type: Structured analysis systems specification tool
Creates, verifies, and documents specifications for real-time systems using the Hatley and Ward/Mellor real-time extensions to Yourdon/DeMarco Structure Analysis
User: Programmers
Mentor Graphics
8500 SW Creekside Place
Beaverton, OR 97005
800-547-7390
VAX/VMS computers, including VAXstation, Apollo workstations
Call for price

Anatool
Type: Prototyping (analysis), development
Implements structured systems analysis on the Macintosh
User: Systems analysts, software engineers, management consultants
Advanced Logical Software
9903 Santa Monica Blvd., Ste. 108
Beverly Hills, CA 90212
213-653-5786
Macintosh
$925
Notes: Anatool checks for consistency, syntax errors, and unfinished work as well as automatically updating the dictionary.

Auditor
Type: Documentation and traceability tool
Tracks requirements in DoD-STD-2167/A design documents
User: Developers
Mentor Graphics
8500 SW Creekside Place
Beaverton, OR 97005
800-547-7390
VAX/VMS mainframes and MicroVAXes
Notes: Interface to Context, Interleaf, and VAXdocument technical publishing systems for documents with integrated text and graphics. Document templates can be customized for company-specific documentation standards.

AutoDraw
Type: Prototyping, development
Draws ER (entity relationship) diagrams from text language statements
User: Systems and data analysts, database analysts
Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$495
Notes: Autodraw is a front-end ER designer; the user enters the text, and the software produces the diagram.

Auto-Mate Plus
Type: Prototyping, development, maintenance
Performs complete analysis, process, and database-design generation; produces detailed programming specifications and database design
User: Systems analysts, database administrators, applications developers
Learmonth & Burchett Mgt. Systems — LBMS Inc.
2900 North Loop W., Ste. 800
Houston, TX 77092
713-682-8530
PC, XT, AT, MS-DOS, PS/2
$7,500 base price plus options
Notes: Also available is an integrated companion product that structures, designs, and selects which system(s) to buy.

B

Bachman/Re-engineering Product Set
Type: Prototyping, development, maintenance through reverse engineering
Allows reverse engineering of existing code and database designs into a logical data model
User: Database administrators, data processing professionals
Bachman Information Systems Inc.
Four Cambridge Center
Cambridge, MA 02142-1401
617-354-1414
Compaq 386, 386/20, PS/2, Model 80
$25,000
Notes: The Product Set allows migration of existing systems through reverse engineering. Currently available are the Data Analyst, Administrator, File Administrator, and Workstation Manager; a full product set is scheduled for release over the next two years.

Brackets
Type: Development
Supports structured design and coding; built-in Cobol code generator
User: Programmers, systems analysts
Optima Inc.
1300 Woodfield Rd., Ste. 400
Schaumburg, IL 60173
800-633-6303
PC, XT, AT, MS-DOS, PS/2
$475
Notes: Brackets interfaces with DesignMachine and DesignVision.

C

C and Pascal Cross Development Tools
Type: Development, maintenance
Optimizing C and Pascal cross compilers

Source: 1988 CASE Industry Directory. *Reprinted by permission of the CASE Consulting Group, Inc.*

User: Developers, programmers
Microtec Research Inc.
3930 Freedom Circle, #101
Santa Clara, CA 95054
408-733-2919
PC compatible
Notes: The product offers manufacturer compatible relocatable macro cross assemblers, linkers, loaders, and high-level language debuggers and supports microprocessors made by 22 different vendors.

CASE 2000 (Design Aid/Data Model Option/Real Time Option/LCM)
Type: Prototyping, development
An integrated structured analysis, design, and documentation tool supported by the LCM methodology manager
User: Analysts, designers, project managers
Nastec Corp.
24681 Northwestern Hwy.
Southfield, MI 48075
800-872-8296
PC compatible
CASE 2000 $6,900, Design Aid $1,500, Data Model Option $1,500, Real Time Option/LCM $2,700
Notes: CASE 2000 interfaces to Ventura publisher, the code generator to Telon plus and the bridge to Softool CCC and Chen's ER Package. It will import/export to other dictionaries via ASCII files.

CardTools
Type: Prototyping, development, maintenance
Designs real-time embedded systems
User: Designers, programmers, systems analysts
Ready Systems Corp.
449 Sherman Ave.
P.O. Box 61029
Palo Alto, CA 94306
415-326-2950
PC compatible
$10,000 base price
Notes: This integrated toolset also addresses documentation and maintenance problems.

CodePilot
Type: CASE tool
Designers
Set of four modules: Control Maps structures diagrams from source code; Object List provides an alphabetized list of program's functions; Navigator provides a way to travel among the functions of the control map screen; Zoom allows instant viewing of functions or files with access to preferred editor.
User: Developers, programmers using C
Train of Thought Inc.
1115 Cleveland St.
Evanston, IL 60202
312-475-7766
800-882-1188
PC or compatibles with 640K RAM
$399
Notes: CodePilot allows software developers to weave between design and code by generating hierarchical charts of the functions in existing software programs. These control maps show the program design while serving as indexes into the source code.

CoFac
Type: Prototyping, development
Produces fully tested and optimized ANSI Cobol programs for PC or mainframe computer
User: Developers, programmers
The Coding Factory Inc.
Brookwood II
45 Knightsbridge Rd.
Piscataway, NJ 08854
201-981-0100
PC compatible
$2,000
Notes: CoFac produces programs with an interactive, menu driven, rule-based interface.

Colonel/Captain
Type: Prototyping, development
A PC resident tool for medium to large database-management systems
User: Developers, systems analysts
Parker Shannon Associates
121 B Tremont St.
Boston, MA 02135
617-787-7842
PC, XT, AT
$10,000 base price
Notes: Colonel produces databases for systems such as IMS/VS, IDMS, IDMS/R, DB2, and VAX Rdb/MS. Auto code generation and documentation are included with Captain option.

Corvet
Type: Prototyping, maintenance, analysis
General CASE tool
User: Programmers
Analysts International Corp.
354 Eisenhower Pkwy.
Livingston, NJ 07039
201-535-9844
PC, XT, AT
Call for price
Notes: Corvet is a programmer's productivity tool.

CorVision
Type: Prototyping, maintenance
Automates code generation and software maintenance from design specifications
User: Data processing professionals
Cortex Corp.
138 Technology Dr.
Waltham, MA 02154
617-894-7000
PC, AT, MS-DOS 3.1 or later; VAX/VMS
$55,000 to $163,000
Notes: The IBM PC is used as a graphics front end only; the program runs on VAX/VMS. Specifications can be created through a VT compatible terminal or IBM PC graphics workstation.

Customizer
Type: Prototyping, development, analysis, design
Helps users shape Excelerator or Excelerator RTS to their own internal standards
User: Information managers, corporate standards group
Index Technology Corp.
One Main St.
Cambridge, MA 02142
617-494-8200
MS-DOS
$12,500
Notes: Customizer works with Excelerator RTS.

D

DDS Link
Type: Development
Creates data dictionary language for target dictionary systems from ER designer files
User: Systems designers
Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$995
Notes: The program is similar to SchemaGen, except DDS Link is targeted for use with the data dictionary.

DFDdraw/SCdraw
Type: Development
Produces graphic diagrams; DFDdraw draws data flow diagrams, SCdraw draws structure charts
User: Analysts, designers
McDonnell Douglas, ISC
P.O. Box 516, Dept. L863
Bldg. 280-L2
St. Louis, MO 63166
800-325-1087
PC, XT, AT, PS/2
$600 each
Notes: DFDdraw and SCdraw support Stradis and other structured technologies (Gane & Sarson, Chen, Bachman, Constantine). The product can also be used for reverse engineering.

Data-Station CAD/Data-Station Dictionary

Type: Prototyping, development
Creates SQL, Cobol, and Ada data definitions
User: Systems analysts, system designers, database administrators, database designers
Charles River Development
483 Beacon St.
Boston, MA 02115
617-424-1820
PC compatible
Call for price
Notes: Data-Station CAD is based on autoCAD.

Deft
Type: Maintenance, design
Structured systems analysis, database design, documentation
User: Analysts, designers, programmers, DBAs
Deft
557 Dixon Rd., Ste. 110
Rexdale, Ont., Canada M9W 1H7
416-249-2246
Macintosh for diagramming/VAX for gateway to relational databases
June 1985; August 1988 for Version 3.1
$9,000
Notes: Deft uses a Macintosh foundation and can be used as a stand-alone product without the VAX component. The new version will have the ability to reverse engineer VAX based databases into Mac Editors.

Depictor, ADR/Datacom, Ideal-Escort, ADR-Look, ADR-Adlib
Type: Development, maintenance
Supports the analysis and design of databases and applications
User: DP/MIS
Applied Data Research Inc.
Rte. 206 and Orchard Rd., CN-8
Princeton, NJ 08543-0008
201-874-9000
PC compatible
$24,500 base price
Notes: These products translate specifications into normalized Datacom/DB database designs.

Design
Type: Prototyping, development
Creates graphic representations illustrating concepts and relationships
User: Analysts, developers, designers,
Meta Software Corp.
55 Wheeler St.
Cambridge, MA 02140
617-576-6920
Macintosh 512K, Plus, SE, II, IBM PC
$250 Macintosh version, $350 PC version
Notes: Design OADS (open architecture development system) sells for $5,000, and Design IDEF sells for $2,000.

Design 1 PC
Type: Prototyping, development
Allows the completing of mainframe applications through unit testing on the IBM PC
User: Analysts, developers, programmers, project managers
Arthur Anderson
33 W Monroe
Chicago, IL 60603
312-580-0033
PC, XT, AT, PS/2

DesignAid
Type: Prototyping, development, maintenance
A structured design and analysis workbench
User: Systems analysts, software engineers
Nastec Corp.
24681 Northwestern Hwy.
Southfield, MI 48075
313-353-3300
MS-DOS, PC-DOS, Version 2.1 or later
$6,900, quantity discounts available
Notes: DesignAid interfaces to various third party products; other production management tools are integrated with the product. Running on a LAN, DesignAid is available with data modeling and real-time systems for $1,500 each.

Designer
Type: Structured design software design tool
Creates and verifies an architectural model of a software system using structured design
User: Designers
Mentor Graphics
8500 SW Creekside Place
Beaverton, OR 97005
800-547-7390
VAX/VMS line of computers including VAXstations, Apollo workstations
Call for price
Notes: Product allows automatic transformation from data flow diagrams to initial software design model as well as reverse engineering of C, Fortran, and all VMS-compiled languages including Ada and Pascal.

DesignMachine
Type: Prototyping, development, maintenance
Automates requirements definition and logical database design
User: Designers, systems analysts, database administrators
Optima Inc.
1300 Woodfield Rd., Ste. 400
Schaumburg, IL 60173
800-633-6303
PC compatible
$10,000
Notes: Formerly produced by Ken Orr & Assoc., DesignMachine now has a logical-database design.

DesignVision
Type: Development
A graphics diagramming facility
User: Designers, programmers, systems analysts
Optima Inc.
1300 Woodfield Rd., Ste. 400
Schaumburg, IL 60173
800-633-6303
PC, XT, AT, PS/2, MS-DOS running Microsoft Windows
$7,500
Notes: DesignVision interfaces with DesignMachine and DesignVision and supports data flow diagrams, structure charts, organizational charts, and ER diagrams. Including a data dictionary, this product supports the creation of a data definition language.

The Developer
Type: Prototyping, development, maintenance
An analysis and design assistant program
User: Analysts, designers, project and database managers
ASYST Technologies Inc.
1080 Beaver Hall Hill, Ste. 1400
Montreal, Que., Canada H2Z 1S8
514-871-0108
800-361-3673
PC, XT, AT
$4,900

Documentation
Type: Source code documentation compiler
Builds cross-indexed document libraries from documentation contained in source code modules or any text file
User: Programmers, project managers
NWP Intelligent Solutions Inc.
P.O. Box 20478
Bloomington, MN 55420
612-884-5860
PC or compatible with 640K RAM
$159.95 for network version, bulk rate discount; $119.95 for single user, personal

E

EPOS
Type: Development, maintenance
An Aerospace/defense, complete life-cycle support environment
User: Project managers
SPS
14 East 38th St., 14th Floor
New York, NY 10016
212-686-3790
AT and compatibles, Intel 286
$8,500
Notes: EPOS contains an Ada code generator and supports 2167A ported to Sun and Apollo platforms.

ER-Designer
Type: Prototyping, development, maintenance

Allows the user to create and manipulate ER diagrams
User: Systems analysts, data analysts, database designers
Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$495 base price
Notes: ER-Designer can be used with most major database software.

Eiffel/Cepage
Type: Prototyping, development, maintenance
A language and environment for software development
User: Developers, programmers
Interactive Software Engineering
270 Storke Rd., Ste. 7
Goleta, CA 93117
805-685-1006
PC compatible
Eiffel $2,995, Cepage $1,995
Notes: This product applies the concepts of object-oriented design and generates C code. Cepage is a document generator that works in conjunction with Eiffel.

Envision
Type: Prototyping, development
An analysis and design tool for software production and maintenance
User: Designers, systems analysts
Future Tech Systems Inc.
724 W Hi-Crest Dr.
Auburn, WA 98001
206-939-7552
PC compatible
$7,500

Excelerator
Type: Prototyping, development, analysis, design
Automates the analysis and design of information systems
User: Developers, systems analysts, project leaders
Index Technology Corp.
One Main St.
Cambridge, MA 02142
617-494-8200
MS-DOS
$8,400

F

Foundation (Method 1, Design 1, Install 1)
Type: Prototyping, development, maintenance
Plans, designs, and implements processing systems in the IBM environment
User: Systems analysts, software engineers, designers
Arthur Andersen
33 W Monroe
Chicago, IL 60603
312-580-0033
PC, XT, AT, PS/2
Design 1 PC to be released first quarter 1989
Method 1 is $50,000 for an on-site license. Design 1 is $7,000 base with discounts for multiple PCs.

Foundation/Vista
Type: Prototyping, development
Analysis and design for the Tandem Pathway environment
User: Analysts, designers, project managers
Menlo Business Systems
The Menlo Bldg.
201 Main St.
Los Altos, CA 94022
415-948-7920
Tandem, Macintosh
Call for price

Foundry
Type: Prototyping, development
Allows the user to create software engineering environments and to create interfaces to other tools
User: Methodologists
The Cadware Group Ltd.
869 Whalley Ave.
New Haven, CT 06515
203-397-2908
PC, XT, AT, PS/2
$8,500
Notes: Foundry includes Rule Tool.

G

GPSM
Type: Prototyping, development, maintenance
Graphically programs simulation models
User: Analysts, developers
Information Research Associates
911 West 29th St.
Austin, TX 78705
512-474-4526
PC compatible
$17,500 base price
Notes: GPSM addresses the product definition phase of the product life cycle.

Gamma
Type: Prototyping, development
Uses information engineering and diagramming methods to define applications systems
User: Developers, systems analysts
KnowledgeWare Inc.
3340 Peachtree Rd. NE, Ste. 2900
Atlanta, GA 30026
404-231-8575
PC compatible
Call for price

Generator II
Type: CASE tool
Designers
Builds structured logic diagrams describing systems, and generates compilable source code in Pascal, C, or Cobol directly from diagram
User: Developers, programmers,
N-TECH
P.O. Box 1162
Golden, CO 80402
303-278-7539
PC compatible
$249

I

IDMS/Architect
Type: Prototyping, development, maintenance
Systems analysis, database design, logical data process design
User: Systems analysts, database analysts, database administrators
Cullinet Software
400 Blue Hill Dr.
Westwood, MA 02090
617-329-7700
PC, XT, AT, MS-DOS, Version 3.2, PS/2 Model 60 and later
$8,000, volume discounts available
Notes: IDMS/Architect performs entity relationship modeling, data flow diagramming, logical data structuring, structure analysis, and finite state machine concept.

IRMA
Type: Prototyping, development
Integrates planning and information design with overall business strategy
User: DP/MIS
Arthur D. Little Inc.
Acorn Park
Cambridge, MA 02140
617-864-5770
PC, XT, AT
Call for price

Information Engineering Facility
Type: Prototyping, development, maintenance (one toolkit for each, sold together or separately)
User: Programmers, systems analysts, consultants, database designers
Texas Instruments
P.O. Box 869305, M.S. 8474

Plano, TX 75086
214-575-4404
AT, MS-DOS
$13,900 for three toolkits
Notes: Used for the entire development life cycle and code production on mainframe, this product is based on the James Martin information engineering methodology. For full productivity the user needs to generate Cobol code on a mainframe, but the tools are also useful for those who hand code.

Information Engineering Workbench
Type: Prototyping, development
Defines applications systems using information engineering and diagramming methods
User: Developers, systems analysts
KnowledgeWare Inc.
3340 Peachtree Rd. NE, Ste. 2900
Atlanta, GA 30026
404-231-8575
PC compatible
Call for price
Notes: This product uses a Ventura-like interface.

InfoModel
Type: Development
An integrated toolkit for data and process modeling
User: Programmers, systems analysts
InfoModel Inc.
137 West 25th St., Ste. 508
New York, NY 10001
212-627-3220
PC, XT, AT
Call for price

Informix
Type: Development
Database management and application building software for a wide variety of systems
User: Analysts, developers, programmers
Informix Software Inc.
4100 Bohannon Dr.
Menlo Park, CA 94025
415-322-4100
MS-DOS
Call for price
Notes: Informix supports SQL.

iPAT 286, iPAT 86/88, iPAT CORE
Type: Prototyping, development, maintenance
Monitors the execution of real or protected mode software in real time
User: Analysts, developers
Intel Corp.
W393,3065 Bowers Ave.
Santa Clara, CA 95051
800-451-2703
PC, XT, AT, DOS
$3,995 for iPAT 286, $2,995 for iPAT 86/88, $4,995 for iPAT CORE
Notes: This software replaces the microprocessor in the prototype system and uses HLL symbols but does not require an incircuit emulator. iPAT CORE is required in order to use iPAT 286 or iPAT 86/88.

L

Life-Cycle Productivity System
Type: Prototyping, development, maintenance, debugging
A set of integrated tools for all phases of development and maintenance
User: Analysts, designers, programmers
American Management Systems
1777 N Kent St.
Arlington, VA 22209
703-841-6000
PC compatible
$6,500 to $16,000

M

MTR
Type: Prototyping, development
Minimum text representation universal method of defining logical processes
User: Developers, designers
Space Ltd.
Coach House Complex
Old St., Upton Upon Severn
Worcestershire, England WR8 ONS
06846-3626
PC, XT, AT
Call for price

MacBubbles
Type: Development
Supports the process of Yourdon/DeMarco structured analysis on the Macintosh
User: Developers
StarSys Inc.
11113 Norlee Dr.
Silver Spring, MD 20902-3619
301-946-0522
Macintosh Plus, SE
$780

MacDesigner
Type: Prototyping, development
Designs and documents software systems on the Mac
User: Developers, programmers
Excel Software
P.O. Box 1414
Marshalltown, IA 50158
515-752-5359
Macintosh 512K, Plus, SE, II
$498
Notes: MacDesigner is customizable; it creates structure charts from child lists built from C, Pascal, Fortran, Cobol, or Modula-2 source code.

Maestro
Type: Prototyping, development, maintenance
Allows user to manage software development; assists in maintenance and reengineering as well as full life cycle automation
User: Data processors
Softlab Inc.
188 The Embarcadero Bayside Plaza,
7th Floor
San Francisco, CA 94105
415-957-9175
PC, XT, AT, MS-DOS, PS/2
Call for price
Notes: Maestro is language, methodology, and target host independent.

managerView
Type: Development
An integrated set of software tools to support computer-aided data and information engineering
User: Developers
Manager Software Products Inc.
131 Hartwell Ave.
Lexington, MA 02173-3126
617-863-5800
PC compatible
Call for price
Notes: managerView uses a PC-to-mainframe (DB2, IMS, CICS) link.

Math Advantage
Type: Development
Software development tools that can be retargeted to architecture of custom high-speed processors
User: Developers
Quantitative Technology Corp.
8700 SW Creekside Pl., Ste. D
Beaverton, OR 97005
503-626-3081
PC compatible
$495 base price

Matrix X
Type: Development
Used for engineering analysis and control system design, non-linear dynamic system simulation from graphical block diagrams
User: Developers of control systems or simulation of dynamic systems
Integrated Systems Inc.
2500 Mission College Blvd.
Santa Clara, CA 95054
408-980-1500
MS-DOS

(continued from page 63)

$995 base system; $4,485 complete system

Notes: Several extra modules are available: Block diagramming simulation package is $2,400, and System ID (reverse engineering) is $995.

MethodManager (managerVIEW)

Type: A data flow diagramming, prototyping, development, and screen painting tool for prototyping (analysis and design), development

User: Systems analysts, data administration groups, applications people

Manager Software Products Inc.
131 Hartwell Ave.
Lexington, MA 02173-3126
617-863-5800
PC compatible
$5,000

Notes: This product is systems-dependent upon the data manager repository on a mainframe; it can upload to a mainframe repository or download and display graphically on a PC. ManagerVIEW is the PC implementation of MethodManager.

MicroStep

Type: Development

Generates executable C programs on PCs from graphically defined specifications

User: Developers, systems analysts

Syscorp International Inc.
9420 Research Blvd., Ste. 200
Austin, TX 78759
512-338-0591
PC, XT with 286 cards, any 286, or 386 with DOS
Call for price

Notes: MicroStep can import/export to dBASE III.

Model

Type: Prototyping, development, maintenance

An automatic software analysis, design, and development system for PL/I, C, and Ada

User: Programmers, designers, systems analysts,

Computer Command & Control
2401 Walnut St., Ste. 402
Philadelphia, PA 19103
215-854-0555
PC, AT
$25,000 per workstation

Model 204/Workshop, 204/PC, 204/Access, 204/Developer

Type: Prototyping, development

An integrated 4GL and relational database management system

User: Database administrators, developers, systems analysts

Computer Corporation of America
Four Cambridge Center
Cambridge, MA 02142
617-492-8860
PC compatible
Call for price

Multi/CAM

Type: Prototyping, development, maintenance

Produces systems development methodologies

User: Analysts, programmers, project managers

AGS Management Systems Inc.
880 First Ave.
King of Prussia, PA 19406
215-265-1550
PC, XT, AT
$97,000 with five workstations

Notes: Multi/CAM supports SDM structured and SDM standard methodologies.

N

NaviText Sam

Type: Prototyping, development

Creates user interfaces following the Smith and Mosier guidelines

User: Developers, systems analysts

Northern Lights Software Corp.
1407 Beacon St., Ste. 7
Brookline, MA 02146
617-566-8500
PC compatible
$295

Notes: NaviText Sam uses HyperText and information management methods and contains 944 guidelines on data entry and display.

Netron/Cap Development Center

Type: Prototyping, development, maintenance

A software development tool for Cobol applications

User: Programmers, systems analysts

Netron Inc.
99 St. Regis Crescent N
Downsview, Ont., Canada M3J 1Y9
416-636-8333
PC, XT, AT
Call for price

Notes: Most users interface with mainframe VAX or Wang systems.

Normalizer

Type: Development

Normalizes data relations to Third Normal Form

User: Systems and data analysts

Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$1,995

Notes: Normalizer can be used as a standalone program or can interface with ER-designer.

O

OPAL

Type: Development

An object-oriented programming language for data definition and manipulation

User: Designers, programmers

Servio Logic Corp.
15025 SW Koll Pkwy., 1A
Beaverton, OR 97006
503-644-4242
PC compatible
$395

P

PAGE

Type: Prototyping, development, maintenance

A multiuser modeling environment

User: Designers, developers, programmers

Inforem Ltd.
40 Uxbridge Rd.
London, England W52BS
01-567-0575
PC compatible
6,000

PAWS

Type: Prototyping, development, maintenance

A performance analysts workbench system

User: Analysts, developers

Information Research Associates
911 West 29th St.
Austin, TX 78705
512-474-4526
PC compatible
$17,500+

Notes: PAWS addresses the product definition phase of the product life cycle; an event-driven simulator provides a dynamic map of system behavior.

PC PRISM

Type: Prototyping, development

Automates the strategic planning and organization of projects

User: Strategic planners, information managers

Index Technology Corp.

One Main St.
Cambridge, MA 02142
617-494-8200
MS-DOS, Version 3.0 or later
$5,000

PDL/81
Type: Prototyping, development
A text-oriented program development tool
User: Developers, programmers
Caine, Farber and Gordon
1010 E Union St.
Pasadena, CA 91106
814-449-3070
PC compatible
Call for price
Notes: PDL/81 offers object-oriented, top down, successive refinement methodologies.

POSE (picture oriented software engineering)
Type: Design, analysis
Nine modules that address planning, design, and analysis phases of application software development
User: Analysts, designers, planners
Computer Systems Advisers Inc.
50 Tice Blvd.
Woodcliff Lake, NJ 07675
201-391-6500
800-537-4262
PC, XT, AT, PS/2, or compatibles; DOS 3.0 or later
$295 per module, $885 for set of four
POSE supports techniques such as Data Modeling, Dataflow and Decomposition Diagramming, Program Structure Charting, and Logical Access Path Analysis.

P-Tools
Type: Prototyping, development, maintenance
A variety of software tools ranging from network-based software control systems to programmer-oriented text editors
User: Analysts, programmers, developers, database administrators
Phoenix Technologies Ltd.
320 Norwood Park S
Norwood, MA 02062
617-769-7020
PC compatible
$145 to $495
Notes: Psource is a DBMS-based source-object control system; PforeCe++ are object-oriented libraries for C++ and C; Pasm is a MASM-compatible Intel family assembler; Pmate is a text processor with a built in macro language; Pfinish is a profiler; Pfixplus is a menu-driven, configurable windows symbolic debugger; Plinkplus is an overlay linker supporting up to 30,000 overlays; Pdisk is a disk backup/restore utility; and Pre-C is a C language syntax checker.

Pacbase
Type: Prototyping, development, maintenance
Automates the full life cycle
User: MIS
CGI Systems Inc.
P.O. Box 1645
1 Blue Hill Drive Plaza
Pearl River, NY 10965
914-735-5030
PC, DEC, Honeywell
$150 to $40,000 mini
Notes: Pacbase produces Cobol programs, including error messages and help screens; it can work from a PC but only in concert with a mainframe.

PC/Focus
Type: Prototyping, development, maintenance
A 4GL/DBMS application development environment
User: Developers, systems analysts
Information Builders Inc.
1250 Broadway
New York, NY 10001
212-736-4433
PC compatible
Call for price
Notes: PC/Focus offers reverse engineering capability.

PC-Magec, WS-Magec, Magec-Excelerator Interface
Type: Prototyping, development, maintenance
A dictionary-driven Cobol application development system
User: Analysts, developers, programmers
Magec Software
P.O. Box 260319
Plano, TX 75026
214-248-0823
214-307-8505
PC compatible
$9,000 for PC-Magec, $1,000 for WS-Magec. Call for price on Excelerator interface.
Notes: This product produces on-line and batch applications.

PolyDoc (The Source Code Documentation Compiler)
Type: Development, maintenance
Assembles documents directly from source code; scans source files and extracts comments and code segments programmer embeds; creates a project documentation library
User: Programmers, project managers,-maintenance personnel
Polytron
1700 NW 167th Place
Beaverton, OR 97006
503-645-1150
800-547-4000
MS-DOS
$149 for network; $450 for five stations

PolyLibrarian
Type: Maintenance
Creates libraries from related object modules
User: Analysts, project managers
Polytron
1700 NW 167th Place
Beaverton, OR 97006
503-645-1150
800-547-4000
MS-DOS
$149 to $297 for five user network
Notes: The linker selects only needed modules, and it has the ability to organize, modify, and reconstruct existing libraries of object code. Compatible with the MS-DOS linker, Plink 86, and others, PolyLibrarian works with compilers and assemblers that produce object modules, and it supports Microsoft format libraries and Intel. This product is integrated with PolyMake for faster builds.

PolyMake
Type: Development (but can also be used for maintenance)
Automatically rebuilds a software system when any module changes
User: Programmers
Polytron Corp.
1815 NW 169th Place
Beaverton, OR 97006
503-645-1150
800-547-4000
PC, MS-DOS
$149 single, $450 five user network
Notes: PolyMake automatically recognizes PVCS and PolyLibrarian time and date stamps.

Polytron Version Control System (PVCS)
Type: Development, maintenance
Simplifies configuration management; maintains a history of revision to a source document; allows prior versions to be recreated; source document can be any file
User: Programmers, managers, work group tool
Polytron Corp.
1700 NW 167th Place

(continued from page 66)
Beaverton, OR 97006
503-645-1150
800-547-4000
PC, MS-DOS, Macintosh/MPW (Macintosh Programmers Workshop)
Personal $149, Corporate $395, Network $995 for five stations (call for larger LAN)
Notes: PVCS is also available on VAX-VMS and Sun Unix; it uses the same archive file format and user interface on all operating systems.

Power
Type: Management
Configuration management, project management tools
User: Designers, managers, systems analysts
Expertware Inc.
3235 Kifer Rd., Ste. 220
Santa Clara, CA 95051-0804
408-746-0706
PC, XT, AT
$9,500

PowerTools
Type: Prototyping, development, maintenance
A package for the analysis, design, and implementation of complex real-time systems
User: Developers, programmers, systems analysts
Iconix Software Engineering Inc.
1037 Third St., Ste. 105
Santa Monica, CA 90403
213-458-0092
Macintosh, 512, Plus, SE, II, PC compatible
$4,495 base price
Notes: Available are these accessory packages: FreeFlow ($1,895), Power PDL ($995), SmartChart ($1,495), FastTask ($1,595) and CoCoPro ($495).

Prodoc
Type: Prototyping, development, maintenance
An integrated analysis, design, code generation, and maintenance system
User: Developers, systems analysts
Scandura Intelligent Systems
Division of Intelligent Micro Systems
1249 Greentree Ln.
Narberth, PA 19072
215-664-1207
PC, XT, AT
$2,400 to $5,600 (Designer/Generator $2,400, Prototyper $2,400, Developer $4,200 to $5,600, Translator $1,800 to $2,200)
Notes: Based on the Scandura FlowForm windowing system, Prodoc supports reverse engineering and uses an English-like syntax.

Project Workbench
Type: Prototyping, development, maintenance
Allows the user to plan and control projects from start to finish
User: Designers, engineers, managers, database personnel, program managers
Applied Business Technology
361 Broadway
New York, NY 10013
212-219-8945
IBM, DEC, Wang, MS-DOS
$1,150, volume discounts available
Notes: Project Workbench plans, revises, and tracks projects from start to finish.

ProKit*Analyst
Type: Development
Used for the analysis phase of systems development
User: Systems analysts
McDonnel Douglas, ISC
P.O. Box 516, Dept. L863
Bldg. 280-L2
St. Louis, MO 63166
314-232-5715
800-325-1087
PC, XT, AT, PS/2
$2,495
Notes: Mainly for those who do not need prototyping or data modeling, ProKit*Analyst is a subset of ProKit*Workbench.

ProKit*Workbench
Type: Prototyping, development, maintenance
A CASE tool to be used during the analysis, design, and development phases of the life cycle
User: Analysts, designers
McDonnel Douglas, ISC
P.O. Box 516, Dept. L863
Bldg. 280-L2
St. Louis, MO 63166
314-232-5715
800-325-1087
PC, XT, AT, PS/2, 3270 PC
$9,200 with volume discounts available
Notes: Containing screen painting and user simulation options, ProKit*Workbench graphically and textually puts information into data dictionary. It interfaces with Telon, and a LAN version is available.

ProMod SA, RT, MD, DC, TMS, ProCap, ProSource, CM
Type: Prototyping, development, maintenance
Supports the full development of the software life cycle
User: Developers, designers, systems analysts
ProMod Inc
23685 Birtcher Dr.
El Toro, CA 92630
714-855-3046
PC, XT, AT
$2,995
Notes: These products support reverse engineering.

Q

Q/Auditor
Type: Development
Evaluates Cobol source code for adherence to Cobol standards
User: Programmers
Eden Systems Corp.
1180 Medical Ct.
Carmel, IN 46032
317-848-9600
MS-DOS, OS/2
Call for price

R

RS1
Type: Data analysis software
Regression and data analysis, mathematical modeling, graphics
User: Scientists, engineers, other technical professionals
BBN Software Products Corp.
10 Fawcett St.
Cambridge, MA 02238
617-873-5000
800-251-1717
VMS, Ultrix, VM/CMS, PC-DOS, Unix
Call for price
Notes: A built-in programming language, RPL allows users to tailor the system's capabilities to their own needs.

RS/Discover
Type: Experimental design software
Plans, conducts, executes, graphs industrial experiments
User: Scientists, engineers
BBN Software Products Corp.
10 Fawcett St.
Cambridge, MA 02238
617-873-5000
800-251-1717
VMS, VM/CMS, Ultrix, Unix
Call for price
Notes: RS/Discover includes RS/Explore and RS/1 in the package.

RS/Explore
Type: Statistical advisory software
Uses statistics to analyze technical data

User: Scientists, engineers, non-statisticians
BBN Software Products Corp.
10 Fawcett St.
Cambridge, MA 02238
617-873-5000
800-251-1717
RS/1 software prior installation; VMS, Ultrix, VM/CMS, Unix
Call for price
RS/Explore is part of the RS/Discover, RS/1 package.

RS/QCA
Type: Statistical quality control software
Gives access to key methods of statistical process control, such as process capability studies, trend analysis, control charts
User: Manufacturing professionals
BBN Software Products Corp.
10 Fawcett St.
Cambridge, MA 02238
617-873-5000
800-251-1717
RS/1 software prior installation; VMS, VM/CMS, Ultrix, PC DOS, Unix
Call for price
Note: The RS/QCA includes RS/1.

Reverse DDS-Link
Type: Development
Draws an ER diagram from data dictionary information
User: Systems analysts, database designers
Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$1,995
Notes: Reverse DDS-Link is intended for redesign work.

S

SAW
Type: Prototyping, development, maintenance
Combines Cadre's Teamwork SA/SD tools with the MicroCASE Software Analyst Workstation
User: Developers, programmers, designers
MicroCASE Inc.
19545 NW Von Neumann Dr.
Beaverton, OR 97006
800-547-4455
PC, AT
Call for price
Notes: SAW provides development tools from design through testing for the microprocessor software designer; an integrated package will be available soon.

SQL*Design Dictionary
Type: Prototyping, development, maintenance
Creates documentation for databases, tables, indexes
User: Systems analysts, database designers
Oracle Corp.
20 Davis Dr.
Belmont, CA 94002
800-345-DBMS
IBM compatibles running MS-DOS, PC-DOS
$1,295 base price
Notes: Other options are pro*Cobol (precompiler), sql*net ($395), and sql*design dictionary ($2,999).

SchemaGen
Type: Development, maintenance
Converts ER diagrams into a target DBMS schema
User: Database designers
Chen and Associates Inc.
4884 Constitution Ave., Ste. 1E
Baton Rouge, LA 70808
504-928-5765
PC compatible
$495 to $4,000

Softcost-R/Softcost-Ada
Type: Special
PC-based, cost estimating packages
User: Systems analysts, accounting departments
Reifer Consultants Inc.
25550 Hawthorne Blvd., Ste. 208
Torrance, CA 90505
213-373-8728
PC compatible
$2,995 to $8,000 (lease)
Notes: This product allows the user to predict the costs and schedules of software products.

Sourceview
Type: Prototyping, development
A Warnier/Orr diagrammer and code formatter
User: Developers, systems analysts
Heartland Systems
1601 Kasold Dr.
Lawrence, KS 66046
913-749-2626
PC, Xenix
Call for price
Notes: Sourceview offers reverse engineering and open architecture.

Structured Architect/Structured Architect Integrator
Type: Prototyping, development
A CASE analysis, design, and documentation tool using the central project database
User: Analysts, designers
Meta Systems
315 Eisenhower Pkwy., Ste. 200
Ann Arbor, MI 48104
313-663-6027
PC compatible
$3,500 for Structured Architect, $15,000 for integrator
Notes: This product offers a reverse engineering service that takes Cobol and creates PSL/PSA statements.

System Architect
Type: Analysis, development, maintenance
User: Designers, programmers, systems analysts
Popkin Software Systems Inc.
111 Prospect St., Ste. 505
Stamford, CT 06901
203-323-3434
Microsoft Windows 286 or 386
$1,395 (special offer)
Notes: The data dictionary and encyclopedia are stored in dBase III Plus format and are available for customized analysis by the end user. A network version is available, and System Architect supports real time.

System Developer
Type: Development
Supports systems analysts and design complex systems
User: Analysts
The Cadware Group Ltd.
869 Whalley Ave.
New Haven, CT 06515
203-397-2908
PC, XT, AT
$3,000
Notes: The technique independent System Developer incorporates former product Design Graphics System.

Sylva Foundry
Type: Workbench for creation of CASE analysis and design tools
Structures CASE environment, creates new techniques, modifies established techniques
User: Software engineers who create software engineering environments
Cadware
869 Whalley Ave.
New Haven, CT 06115
203-397-2908
800-CADWARE
MS-DOS
$3,000 for optional starter kit (for modi-

fying and blending techniques); $2,500 for optional platforms (for dissemination of environments and techniques)

T

T
Type: Development, maintenance
Automatically designs, generates, traces, and documents software test cases
User: Developers
PEI Programming Inc.
4043 State Highway 33
Tinton Falls, NJ 07753
201-918-0110
PC, AT
Call for price

Teamwork
Type: Development
Technical and embedded systems
User: Developers of engineering and scientific applications
Cadre Technologies Inc.
222 Richmond St.
Providence, RI 02903
401-351-5950
OS/2, Unix, VMS, Domain
OS/2 version $4,995; workstation version, $7,500 to $15,000

TCAT
Type: Development, maintenance
Automated software testing tools and services
User: Developers
Software Research Inc.
625 Third St.
San Francisco, CA 94107-1997
415-957-1441
DOS
Call for price
Notes: TCAT offers regression testing and test file generator and supports Cobol, C, Pascal, Basic.

TIP/PC
Type: Planning, analysis
Supports systems planning, database planning, business analysis, requirements definition; a customized version of Excelerator from Index Technologies requiring Excelerator
User: Planners, analysts, systems designers
Technology Information Products
107 Audubon Rd., Bldg. 3
Wakefield, MA 01880
617-246-7720
MS-DOS
$800 (assumes Excelerator already owned)
Notes: TIP/PC supports TIP Proprietary methodologies of TIP Plan and TIP Define.

Teamwork
Type: Development
Technical, embedded systems
User: Developers of engineering and scientific applications
Cadre Technologies Inc.
222 Richmond St.
Providence, RI 02903
401-351-5950
OS/2, Unix, VMS, Domain
OS/2 version, $4,995; workstation version, $7,500 to 15,000

Telon
Type: Prototyping, development, maintenance
Turns design input into Cobol or PL/1 source programs, provides a test facility
User: Developers, systems analysts, programmers
Pansophic Systems Inc.
709 Enterprise Dr.
Oak Brook, IL 60521
312-572-6000
PC, XT, AT, MS-DOS, PS/2
$9,500 to $60,000
Notes: A wide variety of interfaces between Telon and front-end analysis and design CASE tools are available.

V

Visible Analyst, Visible Rules, Visible Dictionary
Type: Development
Interactive graphics analysis and design product
User: Programmers, project managers, systems analysts, systems engineers
Visible Systems Corp.
49 Lexington St.
Newton, MA 02165
617-969-4100
PC, XT, AT
$595 each

VS Cobol Workbench, Cobol/2 Workbench, XL/Interface, Animator
Type: Prototyping, development, maintenance
An integrated set of Cobol compiler and productivity tools
User: Analysts, developers, programmers
Micro Focus Inc.
2465 E Bayshore Rd., Ste. 400
Palo Alto, CA 94303
415-856-6134
PC, XT, AT, PS/2
Call for price
Notes: Excelerator interface converts Excelerator dictionary entries into structured skeletal Cobol.

vsDesigner
Type: Development, maintenance, limited prototyping
Software structured analysis, structured design
User: Systems analysts, software designers, data processors, engineers
Visual Software Inc.
3945 Freedom Circle
Santa Clara, CA 95054
408-988-7575
PC-DOS
$8,500 with quantity discounts
Notes: vsDesigner uses standard methodologies.

vsObjectMaker
Type: Development, maintenance
Allows users to customize object designs and tie them into standard designs
User: Systems analysts, software engineers, (non-industry standard)
Visual Software Inc.
3945 Freedom Circle
Santa Clara, CA 95054
408-988-7575
PC-DOS
$3,995
Notes: Free auto demo/evaluation diskettes are available.

vsSQL
Type: Development, maintenance
An SQL interface to design repository created by vsDesigner
User: Designers, systems analysts
Visual Software Inc.
3945 Freedom Circle
Santa Clara, CA 95054
408-988-7575
PC-DOS
$1,995 with quantity discounts
Notes: vsSQL extracts and imports information into the design repository and interfaces to third party software; open architecture into design database.

W

Windows Systems Architect
Type: Prototyping, development
CASE analysis and design in the Microsoft Windows environment
User: Programmers, systems analysts, database managers
Chelsea Systems Inc.
132 Nassau St., Ste. 1300
New York, NY 10038
212-267-4571
IBM PC running Microsoft Windows
$1,795 basic package, $495 diagramming package, $195 flowcharter
Notes: Windows Sytems Architect is methodology independent.

GLOSSARY

A

Action diagram — A program logic notation combining graphics and text to allow a design to move smoothly from high levels to low levels of detail. Brackets are the basic structuring device and are used in conjunction with pseudocode or compilable program statements.

Artificial intelligence (AI) — A variety of advanced computing approaches, many focused on the notion of symbolic computing, i.e., processing symbols with complex semantics as well as numerical data. Many techniques having their roots in AI have moved into the mainstream of computing, even though the field of AI itself has so far failed to develop as early commercial developers had hoped. Among the concepts tracing their ancestry to AI are: object-oriented systems, expert systems, knowledgebases, heuristic search algorithms, symbolic programming (`a la Lisp), rule-based processing, inference engines, and machine learning. In the context of CASE, AI usually refers to the use of a knowledgebase that contains rules for software design in addition to the objects themselves, and the use of rule-based processing and expert systems to provide more sophisticated automation of the software development process.

Automatic code generation — The ability of a CASE toolset to produce compilable or executable code for a particular target environment based on high-level specifications. At the present time, the meaning of "high level" varies widely depending on the type of software and application domain. In general, more constrained applications, such as on-line business systems approach full code generation, while more complex applications such as transaction processing, process control, and real-time systems achieve less than 70 percent with the remainder coded by hand. Most CASE vendors are working to extend the linkages between front/back-end (code generation) tools.

B

Back-end tools — Tools targeted for the implementation, testing, verification, or reverse engineering of software (in contrast to front-end tools for requirements definition, analysis, and software design).

Baseline — 1. A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures. 2. A configuration identification document or a set of such documents formally designated and fixed at a specific time during a configuration item's life cycle. Baselines, plus approved changes from those baselines, constitute the current configuration identification [IEEE].

C

CASE tool — A software program that provides partial or total automation of a single function within the software life cycle, e.g., a structured analysis or entity-relationship diagram editor, a data dictionary consistency checker, a syntax-directed editor, or a compiler.

Change control — A formal process, often aided by automated tools, of managing the changes in source code from one software version to the next. A change control system defines a check-in/check-out process for approved software modules and limits authorization to make changes to the "official" code. It also provides an archive mechanism that makes it possible to retrieve any previous version of a software system as it evolves.

Code skeleton — The "outer skin" of a software module that defines its interface to the rest of the system. Software skeleton typically includes the module name, parameters, data definitions, package definition (if applicable), and return mechanism. In some cases, some basic logical structures are also included. The skeleton may be executable but will act only as a "dummy" with no functionality until the programmer inserts the processing algorithm by conventional source code editing.

Completeness — The state of a software system in which each baseline requirement is demonstrably met by one or more identifiable design components.

Compliance matrix — A two-dimensional table indicating the cross referencing between requirements and design components in order to establish that all requirements are satisfied by the software design (completeness) and to indicate which modules are critical to the implementation of specific requirements.

Compliance — The degree to which a software system meets the specified requirements (also correctness).

Computer-aided software engineering (CASE) — The creation of software systems using a well-defined design technique and development methodology, supported by computer-based design automation tools.

Concurrency — The simultaneous (or apparently simultaneous) execution of multiple software modules that communicate to satisfy system functional requirements.

Configuration control — The management, often with the aid of automated tools, of the set of related modules that make up a complete software system. Configuration control is typically done at the file level, acting as a counterpart to change control. It also enforces change authorization and check-in/check-out procedures. In addition, a configuration control system often automates the process of converting the set of modules into one, linked executable image (i.e., the "build" process) and maintains the list of included modules and parameters relevant to the configuration process.

Configuration item — Hardware or software, or an aggregation of both, which is designated by the contracting agency for configuration management [DoD]. Consistency checking. Verifying the design elements do not violate any of the rules of the design technique being employed. For example, level balancing of data flow diagrams is a form of consistency checking for the Yourdon-Constantine notation. Control flow diagram (CFD). A graphical notation that expresses the relationship between control algorithms and data processing algorithms in a program.

D

Data dictionary (DD) — A structured listing of the data flows within a program and their breakdown into basic elements. Typically, a variation of BNF is used to indicate the construction of basic elements into composite data flows.

Data flow diagram (DFD) — A graphical notation that expresses the flow and transformation of data within a program. The symbology is essentially that of directed graphs.

Decision table — A method to define all possible alternative responses to input conditions in a program. Decision tables are

Source: 1988 CASE Industry Directory. *Reprinted by permission of the CASE Consulting Group, Inc.*

effective when all possible responses are mutually exclusive.

Decomposition diagram — A graphical notation showing the functional breakdown of a system from the most general, down to a level of detail representing individual program modules.

Diagramming tool — Any CASE tool that allows interactive editing of a graphical design notation on a computer screen. Most diagramming tools also incorporate design rule checking, either interactively or in batch mode, upon invocation by the designer.

DoD-STD-2167 — A formal Department of Defense standard defining the deliverables and the development process to be used in building "mission critical" defense software systems.

E

Entity-relationship (ER) diagrams — A design notation developed by Peter Chen that expresses the data structure of a system in terms of the actual entities of concern in the real system under analysis and the relationships among those entities.

Expert systems — A software system that uses rule processing to deal with incomplete, inaccurate, or contradictory data and to solve problems using heuristics rather than deterministic algorithms. Expert systems are typically composed of two parts: a knowledgebase that contains data and rules and an inference engine that responds to the environment by scanning the rules in the knowledgebase for ones that relate to the current input and taking actions prescribed by the matching rules. Sophisticated search algorithms that limit the range of possible rules that must be considered are crucial elements in expert system technology.

F

Fifth generation (5GL) technology — Computer technologies concerned with developing systems that produce results normally associated with human intelligence such as robotics, natural language processing, automatic theorem proving, knowledge engineering and expert systems [McCLURE].

Formalism — A language and discipline for analyzing a system or part of a system, or specifying a software design. For example, Yourdon, Gane & Sarson, Ward-Mellor, Boeing-Hatley, Constantine, Chen, Bachman, and so on.

Fourth generation (4GL) language — A high-level, non-procedural, end-user-oriented programming language typically tied to an online database management system that allows users to define database query, report, and transaction-oriented applications with fewer statements than required for traditional languages such as Cobol.

Framework — A software system designed to integrate a diverse set of CASE tools into an integrated environment with a consistent user interface. Some frameworks provide a variety of data-management services such as translation among the various tools, version control, and configuration control. Some frameworks can also mediate between the CASE tools and the operating system to provide more portability across a variety of platforms.

Front-end tools — CASE tools that address the requirements definition, analysis, and high-level design of software systems. Most front-end tools tend to combine graphical and textual specification languages and provide automatic checking for adherence to design rules.

Full CASE environment — An integrated environment on one or more computing platforms that provides automation for the entire software life cycle as well as the related management and support activities such as project planning, estimating and management, documentation, configuration control, version control, reusable code library maintenance, team communications, and so forth.

G

Gane-Sarson notation — A structured analysis and design notation developed by Chris Gane and Trish Sarson that is functionally similar to the Yourdon notation.

H

Hatley notation — A design notation, developed by Derek Hatley, for describing the architectural, behavioral, and information processing characteristics of a real-time system.

Heuristics — Decision and transformation criteria based not on deterministic algorithms but rather on an assortment of rules that could apply to a range of possible scenarios, i.e., "rules-of-thumb."

I

Inference engine — The "executing" portion of an expert system that responds to external input by searching for rules and data in an associated knowledgebase that relate to the current state of the system, and producing output and intermediate results as prescribed by the relevant rules.

Information engineering (IE) — An interlocking set of formal techniques in which enterprise models, data models, and process models are built up in a comprehensive knowledgebase and are used to create and maintain data processing systems [KNO].

Internal consistency — 1. No two statements in a document contradict one another; 2. A given term, acronym, or abbreviation means the same thing throughout the document; 3. A given item or concept is referred to by the same name or description throughout the document.

J

Jackson notation — A structured analysis and design notation based on tree structures that describe the structure of data and program modules and also indicate sequence and iterative properties of the system.

K

Knowledgebase — An information repository that contains definitions of the objects that comprise a software system design representation and the relationships among objects as well as the syntactic and process rules that define a correct design within the design methodology in use.

L

Level balancing — Verification of a multi-level data flow diagram to ensure that all inputs and outputs entering and leaving a parent module have corresponding inputs and outputs on the corresponding child diagram.

Life cycle — The series of states that a software system goes through from initial concept through retirement. The traditional "waterfall" model recognizes the states of requirements definition, requirements analysis, system design, implementation, testing, and maintenance. More recently, less linear variations of the waterfall model seem appropriate to some situations, such as business systems that continue to evolve for many years and complex engineering systems that require extensive prototyping before a design can be finalized. These tend to be better described by a "spiral" or iterative life cycle.

GLOSSARY

M

Methodology companion — A CASE tool that provides computerized assistance for a particular software development methodology such as Information Engineering, Data Structured Systems Design, Spectrum, AGS, PMG, DoD-STD-2167, and so forth.

Methodology — A well-defined development process that provides for controlled and orderly progress toward completion of a software system that meets all specified requirements within specified budget and schedule constraints.

Mini-spec(ification) — Used in the Yourdon-Constantine notation to describe the logical and algorithmic content of a basic program module.

O

Object-oriented database — A network database that encapsulates not only data elements, or "objects," but the relationships to other objects and the processing rules unique to each object. Typically, object-oriented databases implement many of the concepts found in object-oriented languages such as Smalltalk: information hiding, inheritance, classes, abstract data types, and messages.

Object-oriented design (OOD) — A design technique that incorporates the concepts of object-oriented programming such as information hiding, inheritance, classes, abstract data types, and messages. OOD is becoming popular in the Ada segment because Ada, the DoD-required programming language for "mission critical" defense systems, is based on object-oriented principles. Buhr notation is one graphical approach to OOD, although it implements only a subset of all OOD principles.

P

Process activation table (PAT) — A tabular specification of conditions that result in the activation of a data transformation process, typically used for real-time systems design.

R

Rapid prototyping — Using CASE tools to successively refine a software design by quickly generating a working prototype and feeding back prototype test results to improve the design specification.

Real-time system — A software system that interacts closely with an external physical environment and that must respond to external physical events in the time frame dictated by the characteristics of the external system.

Repository — An information storage facility. In the context of CASE, repository refers to a central design database that contains all the information relevant to the design and implementation of a software system, including design representations, design rules, and management information.

Requirement — 1. A condition or capability needed by a user to solve a problem or achieve an objective. 2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. The set of all requirements forms the basis for subsequent development of the system or system component [IEEE].

Requirements allocation — The process of partitioning requirements into separate configuration items and associating them with specific design elements within the system architecture.

Requirements management — A rigorous method for establishing, maintaining, and reporting the correspondence between a system's requirements specification and its architecture, components, modules, interfaces, test approaches, and test data throughout the software life cycle.

Reusability — The ability to use an existing software module to satisfy the requirements of a new system. Reusability implies the ability to design generic modules that are easily modified to meet similar but different requirements, and the ability to save and restore a large number of such modules in a widely accessible library and to identify the applicability of specific modules to newly-defined functions (i.e., an indexing facility).

Reverse engineering — The process of transforming existing source code into higher-level design representations automatically or semi-automatically.

S

Screen painter — A CASE tool that allows a designer to define application input, query and edit screens, perform data edit functions and some simple calculations without having to actually write the required code. Typically this is done through a menu-driven session in which the user defines and places the desired fields and fills in table defining data attributes and ranges and computed data definitions.

Software engineering — 1. The systematic approach to the development, operation maintenance, and retirement of software [IEEE]. 2. A discipline for creating software programs that includes a design notation, a design process, and well-defined stages and deliverables throughout the development process.

Software ICs — Software modules that perform generic functions and that are designed in such a way to be easily incorporated into specific software systems. To be generally useful, a software IC must have a carefully design external interface and must be robust enough to adapt to a wide range of specific applications. Object-oriented design shows great promise in making the software IC concept generally useful.

Software life cycle — The sequential stages that a software program or group of programs (system) pass through from initial concept through retirement. The top-down, or waterfall, life cycle model recognizes the phases of requirements definition, analysis, design, implementation, verification and testing, and maintenance. New models, however, such as the rapid prototyping or iterative model, have advantages over the traditional approach in some cases.

Software quality — A planned and systematic pattern of actions necessary to provide adequate confidence that the item or project conforms to established technical requirements [IEEE].

Software workstation — A complete environment including hardware and software whose function is to provide computerized assistance for the production, maintenance and project management of software system [MARTIN].

State transition diagram (STD) — A graphical notation for showing the behavioral characteristics of a software program. State transition diagrams express program behavior as a series of states and their related outputs, along with all the possible transitions between states and the conditions that can cause these transitions.

Statecharts — A variant of state transi-

tion diagrams developed by David Harel to describe the behavioral design of a real-time system. Statecharts incorporate a rich textual/logical language in addition to the typical graphical state diagram notion.

Structure chart (SC) — A graphical notation that shows the structural decomposition of a software system into related modules indicating calling relationships and parameters passed between modules. Structure charts visually show characteristics of a system such as coupling and cohesion and provide a convenient system overview when working at the detailed coding level.

Structured analysis (SA) — A top-down software analysis technique that stresses complete definition of the requirements of a system before the design is initiated. SA usually employs a successive refinement process and graphical notation to express system requirements.

Structured design (SD) — A top-down software design technique that employs successive refinement to define the structure and external specifications of a system using graphical notation.

Synchronization — A mechanism to ensure multiple software modules that respond to or depend on the results of each other will communicate at the appropriate times.

T

Task — A single-threaded module that performs a specific function in a larger system of related modules. The notion of "task" is only useful in a system that has multiple modules executing concurrently.

Technique — A language and discipline for analyzing a system or part of a system, or specifying a software design. For example, Yourdon, Gane & Sarson, Ward-Mellor, Boeing-Hatley, Constantine, Chen, Bachman, etc.

Toolkit — A set of integrated tools that automates a major step in the software development life cycle, such as system analysis, program design or software implementation; or a major functional task, such as software maintenance or configuration control.

Traceability — The ability to show the correspondence between a specific requirement and the project deliverables that satisfy that requirement. The ability to indicate the specific requirement(s) fulfilled by a given project deliverable. When applied to a hierarchical set of design documents, traceability has five elements: 1. The document in question contains or implements all applicable stipulations of the predecessor document. 2. A given term, acronym, or abbreviation means the same thing in the documents. 3. A given term or concept is referred to by the same name or description in the documents. 4. All material in the successor document has its basis in the predecessor document (no untraceable material has been introduced). 5. The documents do not contradict one another [IEEE].

V

Validation — The process of evaluating software at the end of the software development process to ensure compliance with software requirements [DoD].

Verification — 1. The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. 2. The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements [DoD].

W

Ward-Mellor notation — An extension of the Yourdon-Constantine notation, developed by Paul Ward and Stephen Mellor, that incorporates constructs required to model real-time systems.

WarnierOrr diagram — A system decomposition and program structure notation, developed by Jean Warnier and Kenneth Orr, using vertical brackets to group data and logic elements hierarchically.

Workbench — 1. An assembly of integrated sets of tools whose function is to automate the production and maintenance of software systems as well as the software project management activities. CASE toolkits are combined into CASE workbenches to automate tasks across the entire life cycle [McCLURE]. 2. An integrated set of tools that automates the entire spectrum of activities performed in a particular job classification, for example, analysis and design, programming and project management workbenches.

Y

Yourdon-Constantine notation — A top-down, structured design technique using data-flow diagrams, data dictionaries, mini-specs, and structure charts to represent the design process from requirements analysis through program modular design.

References

IEEE "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 729 – 1983.

DoD "Department of Defense Military Standard, DoD-STD-2167," Department of Defense.

MARTIN James and McClure, Carma, "The Latest Look In Programmer Productivity Tools," Business Software Review, May, 1986.

McCLURE Carma, "Proceedings of the Computer-Aided Software Engineering Symposium," Digital Consulting Inc., Fall, 1987, p. J-11.

KNO "Information Engineering: A Management White Paper," KnowledgeWare Inc., 1986.

BIBLIOGRAPHY

A

Ada Information Clearinghouse Newsletter. Newsletter produced every other month by the Ada Joint Program office. 301-731-8894 or 703-685-1477. No charge.

Ada Letters. A bimonthly publication of SIGAda, the ACM special interest group on Ada. SIGAda annual dues: $15 for ACM members; $10 for ACM student members; $37 for non-ACM members; ACM Inc., P.O. Box 12115, Church Str. Sta., New York, NY 10249; 212-869-7440.

Ada Strategies. Monthly newsletter published by Cutter Information Corp. Designed for groups most affected by Ada by providing information and analyses for developing strategies and contract evaluation guidelines; 617-648-8700.

AdaData. International Resource Development Inc. A monthly newsletter covering market trends and commercial developments in Ada software, services, and equipment. 21 Locust Ave., #1C, New Canaan, CT 06840; 203-966-2525. $315.

Adas News. Monthly publication of Research Triangle Institute, Research Triangle Park, NJ. Reporting on the use of and new enhancements to ADAS; 919-541-7044.

Agresti, William W. *Tutorial: New Paradigms for Software Development.* IEEE Computer Society Press, June 1986.

Arthur, L.J. *Measuring Programmer Productivity and Software Quality.* John Wiley and Sons, 1985. A somewhat superficial, but useful, survey of software metrics. Presents measures of the software development process and technical measures of software. Considers both quantitative and qualitative measures.

Arthur, L.J. *Programmer Productivity.* John Wiley and Sons, 1983. An extremely readable treatment of software productivity, this book presents a good qualitative overview of the topic. Presents a concise description of function points and their impact on productivity measurement.

B

Babich, W.A. *Software Configuration Management.* Addison-Wesley, 1986. An abbreviated, yet effective, treatment of pragmatic issues in software configuration management. More readable and current than Bersoff et al., this book covers all important SCM tasks and surveys existing tools.

Barstow, D. *A Perspective on Automatic Programming.* Morgan Kaufman Publishers Inc., 1986.

Barstow, D.R., et al. *Interactive Programming Environments.* McGraw-Hill, 1984. A collection of contributed papers and chapters that describe research efforts in "computer-aided software engineering." Important information for builders of software tools.

Basili, V. *Models and Metrics for Software Management and Engineering.* IEEE Computer Society Press, 1980.

Bersoff, E.; Henderson, V.; and Siegal, S. *Software Configuration Management.* Prentice-Hall, 1980. This book is one of the few that focuses on software configuration management. Although the treatment is somewhat general, a number of excellent case studies are presented; good overall guidelines are proposed, and key terminology and concepts are introduced.

Boar, Bernard H. *Application Prototyping.* John Wiley and Sons, 1985. A discussion of prototyping in a data processing environment. Contains good arguments for prototyping as an alternative to conventional specification.

Boehm, Barry W. *Software Engineering Economics.* Prentice-Hall, 1981. Written by the director of software development for TRW, this book presents a comprehensive treatment of software estimating, project planning, and control. Somewhat mathematical and technical in parts, the text introduces Boehm's CCOMO estimation modeling scheme and provides useful data on software development productivity.

Booch, Grady. *Software Engineering With Ada.* Benjamin/Cummings, 1983.

Brooks, Frederick. *The Mythical Man-Month.* Addison-Wesley, 1975.

Buhr, R.J.A., et al. *Software CAD: A Revolutionary Approach.* Carleton University Dept. of Systems and Computer Engineering, June 1986.

C

Cameron, John. *Tutorial: JSP and JSD — The Jackson Approach to Software Development.* IEEE Computer Society Press, 1983.

Case, Albert F. Information Systems Development: Principles of Software Engineering and CASE. Prentice-Hall, 1986. This book is one of the first to emphasize the use of computer-aided software engineering (CASE) as an approach to information systems development. Emphasizes management issues and presents an overview of important techniques. Contains useful guidelines for the implementation of software engineering and CASE.

CASE Outlook. Monthly industry report on the practical application of CASE, including information on technology, methods, products, and organizational issues, news and events, book reviews, and case studies. Case Consulting Group, 224 S.W. First Ave., Portland, OR 97204; 800-523-2203.

Charette, Robert N. *Software Engineering Environments: Concepts and Technology.* Intertext/McGraw-Hill, 1986.

Cherry, George W. *The Pamela Designer's Handbook.* TASC-AdaGraph Headquarters. This two-volume set on the process abstraction method for embedded large applications (PAMELA) teaches the reader how to use the latest software engineering concepts to specify, design, and develop concurrent or parallel software in the Ada language. It is based on George W. Cherry's 4-day PAMELA seminar.

Chow, T.S. *Software Quality Assurance: A Practical Approach.* IEEE Computer Society Press, 1985.

Computer-Aided Software Engineering — Special Report for the CASE Symposium. James Martin Productivity Series. Digital Consulting Inc., 1987.

Cox, B. *Object-Oriented Programming.* Addison-Wesley, 1986. A detailed treatment of object-oriented design and programming using the Smalltalk/Objective-C point of view. Introduces important concepts such as inheritance, encapsulation, and messages.

Source: 1988 CASE Industry Directory. *Reprinted by permission of the CASE Consulting Group, Inc.*

BIBLIOGRAPHY

D

Dahl, O.; Dijkstra, E.; and Hoare, C. *Structured Programming*. Academic Press, 1972.

Database Newsletter. Published every other month by Database Research Group Inc. Source of specialist information for data management professionals. 617-423-2029.

Database Programming and Design. Miller Freeman Publications, San Francisco, Calif.

Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 1986.

Date, C.J. *An Introduction to Database Systems*, vol. I, 3d edition. Addison-Wesley, 1981.

Date, C.J. *An Introduction to Database Systems*, vol. II, 3d edition. Addison-Wesley, 1983.

Date, C.J. *Relational Database: Selected Writings*. Addison-Wesley, 1986.

DeMarco, Tom. *Controlling Software Projects*. Yourdon Press, 1982.

DeMarco, Tom. *Structured Analysis and System Specification*. Yourdon Press, 1978.

Deming, W. Edwards. *Out of Crisis*. Cambridge: MIT Center for Advanced Engineering Study, 1986.

Department of Defense. *DOD-STD-2167: Software Development*. Naval Publications and Forms Center, Philadelphia, Penn.

Department of Defense. *DOD-STD-2168: Software Quality Evaluation*. Naval Publications and Forms Center, Philadelphia, Penn.

Department of Defense. *MIL-Q-9858A: Quality Program Requirements*. Naval Publications and Forms Center, Philadelphia, Penn.

Department of Defense. *MIL-STD-490A: Specification Practices*. Naval Publications and Forms Center, Philadelphia, Penn.

Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, 1976.

E

Electronic Design Interchange Format Steering Committee. *EDIF Electronic Design Interchange Format*, Version 2.0.0, Reference Manual. Electronic Industries Assoc., 1987.

Evans, M.W. and Marciniak, J.J. *Software Quality Assurance and Management*. John Wiley and Sons, 1986. A management-oriented view that contains good information on software quality evaluation, metrics, product issues, and data control. Makes use of the latest IEEE SQA standards.

F

Fairley, R. *Software Engineering Concepts*. McGraw-Hill, 1985. A lifecycle treatment of the software engineering process. Presents a worthwhile overview of important aspects of the technology. Contains a good bibliography.

Feigenbaum, E.A. and McCorduck, P. *The Fifth Generation*. Addison-Wesley, 1983.

Foster, C.C. *Real-Time Programming — Neglected Topics*. Addison-Wesley, 1981.

Freeman, Peter. *Tutorial: Software Reusability*. IEEE Computer Society Press, 1987.

G

Gane, Chris. *Computer-Aided Software Engineering: The Methodologies, The Products, The Future*. Rapid System Development Inc., 1988.

Gane, Chris and Sarson, Trish. *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall, 1979.

Glass, R.L. *Real-Time Software*. Prentice-Hall, 1983.

Goldberg, Adele, *The Influence of an Object-Oriented Language on the Programming Environment*. McGraw-Hill, 1984.

Government Source Code Directory. "Source Translation & Optimization, 1987." 150-page directory of over 8,000 programs available in source code from the government. Directory is also available in machine readable form; 617-489-3727.

Gunther, R.C. *Management Methodology for Software Product Engineering*. John Wiley and Sons, 1978. This book provides excellent guidance for managers who must plan and control products in which software is a critical (but not the only) element. The concept of "software as a product" is introduced and the management disciplines to control product planning, development, services, documentation, support, testing, and maintenance are discussed. A number of interesting case studies, guidelines, and outlines are included.

H

Hatley, Derek J. *A Structured Analysis Method for Real-Time Systems*. Fall DECUS US Symposium, 1985.

Hatley, Derek J. and Pirbhai, Imtiaz. *Strategies for Real-Time System Specification*. Dorset House, 1987.

Hatley, Derek J. and Pirbhai, Imtiaz. *Integrated Structured Analysis and Design*. Dorset House.

Howard, Philip C., ed. *System Development*. Applied Computer Research Inc.

Human Factors in Software Development, 2d edition. IEEE Computer Society Press, 1986.

I

IEEE Transaction on Software Engineering, volume SE-3, number 1. January 1977. Note: entire issue devoted to structured analysis.

Info DB. *The Journal for Large Systems Database Users*. Colin J. White Consulting, San Jose, Calif.

J

Jackson, M.A. *Principles of Program Design*. Academic Press, 1975.

Jackson, M.A. *System Development*. Prentice-Hall, 1983. Still another approach to data structure-oriented system design. Jackson offers many intriguing insights into the construction of software-based systems.

Jensen, Randall W. and Tonies, Charles C. *Software Engineering*. Prentice-Hall, 1979.

Jones, Capers. *Tutorial: Programming Productivity-Issues for the Eighties* (2d edition). IEEE Computer Society Press, 1986.

BIBLIOGRAPHY

Jones T.C. *Programming Productivity*. McGraw-Hill, 1986. In this significant extension of his original anthology, Jones presents an in-depth discussion of methods for measuring, interpreting, and improving software engineering productivity and quality. A thorough and pragmatic treatment of the subject.

Jones, T.C., ed. *Programming Productivity: Issues for the Eighties*. IEEE Computer Society Press, 1981. This book is an anthology of important papers that address software "productivity" and quality. Methods for measurement of productivity, life cycle analysis, methodologies, and programming environments are considered. The reader can pick and choose from among many topics.

M

Marca, D. *Applying Software Engineering Principles*. Little, Brown and Company, 1984.

Marca, D. and McGowan, C. *SADT: Structured Analysis and Design Technique*. McGraw-Hill, 1987.

Martin, James. *Application Development Without Programmers*. Prentice-Hall, 1982.

Martin, James. *Recommended Diagramming Standards for Analysts and Programmers*. Prentice-Hall, 1987.

Martin, James. *Technology's Crucible*. Prentice-Hall, 1986.

Martin, James and Finkelstein, C. *Information Engineering, Savant Technical Report 22*. Savant Institute, 1981.

Martin, James and Hershey, E.A. III. *Information Engineering: A Management White Paper*. Prentice-Hall.

Martin, James and McClure, Carma. *Action Diagrams: Clearly Structured Program Design*. Prentice-Hall, 1985.

Martin, James and McClure, Carma. *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall, 1985. Probably the most comprehensive survey of analysis and design notation produced to date. Topics include structure charts, HIPO, Warnier-Orr, N-S diagrams, flowcharts, state transition diagrams, PDL, decision trees/tables, Jackson diagrams, and many others.

Martin, James and McClure, Carma. *Software Maintenance, the Problem and Its Solution*. Prentice-Hall, 1983.

Martin, James and McClure, Carma. *Structured Techniques for Computing*. Prentice-Hall, 1985.

McCabe, Thomas J. *Tutorial: Structured Testing*. IEEE Computer Society Press, October, 1982. A detailed treatment of McCabe's Basis Path Testing technique. Includes discussion of software complexity and its use in test case design.

McClure, Carma. *Managing Software Development and Maintenance*. Van Nostrand Reinhold Company, 1978.

McClure, Carma. *Reducing Cobol Complexity Through Structured Programming*. Van Nostrand Reinhold Company, 1978.

McClure, Carma. *The CASE Technical Report*. Extended Intelligence Inc., 1988.

McMenamin, Stephen M., and Palmer, John F. *Essential Systems Analysis*. Yourdon Press, 1984.

Miller, E. *Automated Tools for Software Engineering*. IEEE Computer Society Press, 1979.

Miller, E. and Howden, W.E. *Software Testing and Validation Techniques*, (2d edition). IEEE Computer Society Press, 1981.

Monk, A., ed. *Fundamentals of Human Computer Interaction*. Academic Press, 1984. Contains invited chapters on many important aspects of human-machine interface design. Combines psychological research with technical issues of machine interaction.

Musa, John; Iannino, Anthony; and Okumoto, Kazuhira. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987. Integrates the theoretical and pragmatic aspects of software reliability measurement — the book is filled with practical, real-world projects.

Myers, Glenford J. *The Art of Software Testing*. John Wiley and Sons, 1979. Still the most widely referenced and quoted book on software testing. Introduces the reader to a variety of black-box and white-box testing techniques.

Myers, Glenford J. *Software Reliability, Principles, and Practices*. John Wiley & Sons, 1976.

Myers, Gordon. *Reliable Software Through Composite Design*. Van Nostrand Reinhold, 1975.

O

Object-Oriented Computing, vol. 1: Concepts. IEEE Computer Society Press, 1988.

Object-Oriented Computing, vol. 2: Implementation. IEEE Computer Society Press, 1988.

Orr, Kenneth. *Structured Requirements Definition*. Ken Orr & Assoc., 1983.

Orr, Kenneth. *Structured Systems Development*. Yourdon Press, 1980.

P

Page-Jones, M. *The Practical Guide to Structured Systems Design*. Yourdon Press, 1980.

Peterson, Gerald. *Tutorial: Object-Oriented Computing, vol. 1: Concepts*. IEEE Computer Society Press, January, 1988.

Peterson, Gerald. *Tutorial: Object-Oriented Computing, Volume 2: Implementations*. IEEE Computer Society Press, January 1988.

Pressman, R.S. *Making Software Engineering Happen*. Prentice-Hall, 1988. The only book currently available to treat the problems associated with the installation of software engineering practice. Defines a "life cycle" for software engineering implementation that includes assessment, education, selection, installation, and evaluation.

Pressman, R.S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1982.

Programming Productivity: Issues for '80s (2d edition). IEEE Computer Society Press, 1986.

R

Ramamoorthy, C.V., et al. *Techniques in Software Quality Assurance*. Teubner Verlag, 1982.

Reifer, Donald J. *Tutorial: Software Management* (3d edition). IEEE Computer Society Press, 1986.

Reifer, Donald J., ed. *Software Management*, 2d edition. IEEE Computer Society Press, 1981. This book is an anthology of important papers on software management. Topics include: planning, organizing, staffing, directing, and controlling software projects. A number of case studies are presented.

Release 1.0. EDventure Holdings Inc., New York, NY. Software and computer technology newsletter; 212-758-3434.

Rhayer, Richard. *Tutorial: Software Engineering Project Management*. IEEE Computer Society Press, 1988.

Rich, C. and Waters, R.C., eds. *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufman Publishers Inc., 1986.

Ross, Ronald G. *Entity Modeling: Techniques and Application*. Data Base Research Group Inc., 1987.

S

Scherkenbach, William W. *The Deming Route to Quality and Productivity: Roadmaps and Roadblocks*. ASQC Quality Press, 1986.

Software Configuration Management. IEEE Computer Society Press, 1980.

Software Cost Estimating & Life-Cycle Control. IEEE Computer Society Press, 1980.

Software Design Strategies (2d edition). IEEE Computer Society Press, 1981.

Software Design Techniques (4th edition). IEEE Computer Society Press, 1983.

Software Development Environments. IEEE Computer Society Press, 1981.

Software Development, New Paradigms. IEEE Computer Society Press, 1986.

Software Engineering Project Management. IEEE Computer Society Press, 1988.

Software Maintenance News. Monthly publication covering software maintenance issues; 718-816-5522.

Software Maintenance. IEEE Computer Society Press, 1983.

Software Management (3d edition). IEEE Computer Society Press, 1986.

Software Quality Assurance: A Practical Approach. IEEE Computer Society Press, 1985.

Software Restructuring. IEEE Computer Society Press, 1986.

Software Reusability. IEEE Computer Society Press, 1987.

Software System Design — Description & Analysis. IEEE Computer Society Press, 1980.

Software Testing & Validation Techniques (2d edition). IEEE Computer Society Press, 1981.

Software, Selected Reprints in (3d edition). IEEE Computer Society Press, 1987.

Sommerville, I., ed. *Software Engineering Environments*. Peter Peregrinus Ltd. London, 1987.

Structured Programming: Integrated Practices. IEEE Computer Society Press, 1981.

Structured Testing. IEEE Computer Society Press, 1982.

System Development. Applied Computer Research Inc. Reports on advances in application development productivity; 602-995-5929.

T

The CASE Report. Monthly newsletter published by Nastec Corp. Contains articles of interest about current CASE technology; 313-353-3300.

The Software Productivity Consortium Quarterly. The Software Productivity Consortium, Reston, VA; 703-648-1880.

V

Vick, C.R. and Ramamoorthy, C.V., ed. *Handbook of Software Engineering*. Van-Nostrand Reinhold Company, 1984.

W

Walton, Mary. *The Deming Management Method*. Dodd, Mead & Company, 1986.

Ward, Paul T. *Systems Development Without Pain*. Yourdon Press, 1985.

Ward, Paul T. and Mellor, Stephen J. *Structured Development for Real-Time Systems, vol. 1: Introduction & Tools*. Yourdon Press, 1985.

Ward, Paul T. and Mellor, Stephen J. *Structured Development for Real-Time Systems, vol. 2: Essential Modeling Techniques*. Yourdon Press, 1985.

Warnier, Jean Dominique. *Logical Construction of Programs*. Van Nostrand-Reinhold Co., 1974.

Warnier, Jean Dominique. *Logical Construction of Systems*. Van Nostrand-Reinhold Co., 1981. A discussion of data structure oriented specification by one of the founding fathers.

Wasserman, Anthony I. *Tutorial: Software Design Techniques* (4th edition). IEEE Computer Society Press, November 1983.

Wasserman, Anthony I. *Tutorial: Software Development Environments*. IEEE Computer Society Press, November 1981.

Weinberg, Gerald M. *Rethinking Systems Analysis and Design*. Little, Brown and Co., 1982.

Weinberg, Gerald M. *An Introduction to General System Thinking*. Wiley Interscience, 1976.

Weinberg, Gerald M. *On the Design of Stable Systems*. Wiley Interscience, 1976.

Winsberg, Paul. *CASE: Getting the Big Picture*. Database Programming & Design. vol. 1, no. 3., March 1988.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

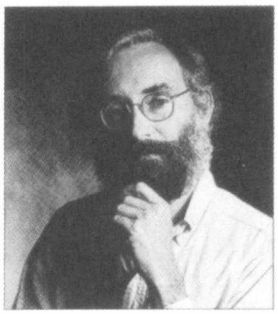
Y

Yourdon, Edward and Constantine, Larry L. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, 2d edition. Prentice-Hall, 1979.

Yourdon, Edward. *Managing the Structured Techniques*. Yourdon Press, 1986.

Yourdon, Edward. *Managing the System Life Cycle*. Yourdon Press, 1982.

CAVEAT VENDOR



Michael Swaine is editor-at-large of DDJ *and coeditor of this issue. He is also author of the recently released* Dr. Dobb's Essential HyperTalk Handbook.

Usually, our creative efforts leave enough fat on the bone to hide what was done when. And even when you can see the skeleton through the skin, the chronology of presentation rarely matches the chronology of composition. Project schedules get worked out from back to front; programs get written from the top down; and magazines get put together from the inside out, from the outside in, or in the order in which the organs arrive at the lab.

Nevertheless, this last page is the last page; this particular bone of contention was the very coccyx of the corpus, the last piece put in place before we ordered Igor to throw the switch. This fact allows me to step alongside you to examine the body on the slab and to reflect on the deed the Doctor has done.

Frankly, we had some doubts about the concept of this issue. Software engineering was important enough, we knew, but in some ways the subject seemed alien to the Doctor's whole reason for being.

It was a very subjective hesitation, but put into words, I think our feelings would read something like this: *DDJ* is about tools for the individual programmer; it's a product of the computer revolution, which we can't help but feel is still going on. *DDJ* is about exploring frontiers, discovering new approaches to problems, even new problems. Software engineering, on the other hand, is all about programming teams, about control and management of complexity, about dealing efficiently with the aspects of computing that we already understand well. If *DDJ* is tinder, software engineering is CO_2.

Ultimately, we realized that distinction was just a runaway metaphor, like the Frankenstein figure I unleashed at the beginning of this piece.

Ultimately, we remembered that the typical *DDJ* reader is the true consumer of software engineering solutions. Regardless who actually buys them, software engineering tools are tools that have to help programmers.

That being the case, let me give a word of advice to all vendors of software engineering tools: Never underestimate the programmer. We've been providing tools to the best programmers for over a decade, and we've never been criticized for talking over their heads. These people are smart. They know what they need. They are unlikely to buy packaging over value.

Caveat vendor.

Michael Swaine
editor-at-large

Index

21st Century Data, 388
32-bit
  data operations, 537
  software developments, 539
34010 Software Developer's Kit, 703
"386 vs. 030: The Crowded Fast Lane," 7
68030, 7
80286, 61, 537, 659
80386, 7
80386 Instruction Assembly, 534
  initializing, 537
  Intel's, 61
  microprocessor, 7
"80386 Assembler in Forth, An," 533
"80386 Protected Mode Initialization," 537
8086 capability, 61

AAL, 173
  overview of, 173
  programming styles, 175
  implementing, 176
action-chart diagramming, 459
activities, 800
Actor, 43
  desktop, 43
  syntax, 44
  controls and dialog boxes, 46
  with AI, 46
  debugging, 47
Ada, problems, 475
"Ada for Pascal Programmers," 466
Ada-ROM, 521
"Adding Extensions to Lisp," 529
Addison-Wesley, 52, 111, 221
Advanced Micro devices 80C521, 389
Adventure Authoring Language, 173
"Aid to Documenting C, An," 408
AI Ware, 222
AIM Technology, 450
AI, 324
AI-Net 101, 222
Alde Publishing, 521
Aldridge Company, 333
algorithms, 302, 412, 687
  Ratcliff/Obershelp pattern matching, 355
Allegro CL, 220
Allen, Dan, 720, 749
ALS Prolog Macintosh, Version 1.0, 574
Al-Dhelaan, Abdullah, 10
American Association for Artificial Intelligence, 181
Amsterdam, Jonathan, 173, 529
ana-systems, 163
ANS Forth Committee, 564
ANSI standard Version 3.3, 389
Anza Plus/VME, 705
APDA, 31
APL*PLUS II, 705
Apple, 10
Apple Computer's Certified Developer Program, 714
Apple Programmer's and Developer's Association, 31
"Apple's A/UX: Unix for the Rest of Us?," 741
Applied Logic Systems, 332, 574
"Applying Workstation Technology to CASE," 803
architectures, 232, 651
  parallel, 687
arguments, passed, 492
"Arguments and Automatic Variables in Assembly Language," 492
Ariel Corp, 576

Arium Corp., 110
arrays: huge, 155, 557; disk-based, 557
"Artificial Intelligence," 43
Asmflow, 575
assembler routines, 608
assemblers, Forth, 533
assembly-language subroutines, 492
ASYNCH PLUS/4.0, 280
asynchronous system requests, 290
Atkinson, Bill, 749
ATrio Systems, 328
AT&T, 655
Austin Code Works, 398
Australian Forth Symposium, 214
automated software design methods, 780
automatic variables, 492, 493
awk, 92
AWK Programming Language, The, 111
A/UX, 741
A/UX Resource Guide

background only applications, 666
backward chaining, 182
Basic, 359, 608
Basics
  New, 360
  Quick 3.0, 360
  Turbo, 1.1, 360
  Quick 4.0, 361
BASIS, 332
batch mode, 289
BBx, 332
benchmarking compilers, 489
Benchmarks, 8, 732
Berkshire Software Company, The, 221
BEYOND.BAT, 333
bibliography, for Software Engineering Sourcebook, 821
Big Bang Software, 388
Biggerstaff, Ted, 24
Binary B+ Trees, 684
binary executable code, 255
bit mapped, 234
Black & White International, 451,
Blackhan, Greg, 650
Blaise Computing, 280
bldfuncs, 404
Borland International, 158, 333, 545, 630
  Turbo C, 401
  Turbo C, Version 1.5, 397
Boyd, Stowe, 770
BPTPLUS, 110
breakpoint, 62 registers, 64
Bricklin, Dan, 279
Brief, Version 2.01, 218
Brief macro, 674
Brown's Algorithm, 301
Budiansky, Michael, 234
bugs, 311
"Building Software for Portability," 650
bulletin boards, 438
B&C, 111

C, 242, 359, 485, 514
  Borland Turbo C, 397, 401
  Datalight Optimum-C,397, 402
  K & R definition, 5
  Microsoft, 397, 399
  Microsoft, Quick, Turbo, 117
  Objective, 416
  Version 1.5, 397
  Version 5.1, 397
  Version 6.5, 397
  Watcom, 397, 401
C++ language, 450, 563
C/2 Version 1.10, 450
C2PS, 521
C86+, 402
C: A Reference Manual, 95
"C Chest," 91, 24, 200, 255, 311, 376
C compilers, 117
C constructs, 435
C programmer, 404
C Programmer's Association, The, 388

"C Programming," 432, 510, 560, 626, 680
C Programming Language, The, 279, 434, 521
C Programming Language, The, Second Edition, 378
C programming library, 513
C source code, searching, 404
C Tool Set, 681
C Torture Test, 398
California 10 PAK, 281
California Software Products, 281
Canon Cat, The, 99
Capps, Steven, 52
Carr, Robert, 231
CASE tools, 805
  workstation, 804,
CASE, 770, 778, 793
CCSL, 575
CDEF Development Files, 150
CDEF, 34, 148
CGI, 585, 594
chaining, backward, 182
Chalk, Andrew J., 128, 671
Charm, 279
CheckMate, 108
chipForth 8051, 575
C-INDEX+, 328
Cinnabar Software, 108
Cito 286, 110
classes, 122, 416
  graphics, 419
"Class-ier C," 677
C:Lines/C:Tree, 110
CLOE, 222
CNS, 677
CNS Inc., 389
COBOL 85 for Programmers, 521
Cochran, Dan, 31
CodeView, 161
code, Macintosh, 33
Cogent Software, 705
ColorBoard 1/104, The, 164
command, DIR, 440
common-subexpression elimination, 255
communications, 510
compilers, 651
  benchmarking, 489
  C, 117
Complete Logic Systems, 221, 451
completion routines, 291
composite units, 481
Comprehensive Guide to AI and Expert Systems: Turbo Pascal Edition, 221
Computer Animation Show, The, 164
Computer Innovations C86+, Version 1.10, 397
Computer Innovations, 402
computer-graphic imagery, 585
concurrency control, 125
Concurrent Sciences, Inc., 104
configuration information, hiding, 91
constructs, C, 435
contest, programming, 438
context-sensitive help, 680
control flow, 787
controls, handling, 152
Convenience Plus DOS Front End, 280
Copeland, Ron, 327, 456, 526, 582
Copia International Ltd., 327
copyright, software, 57
Coral Software Corp., 163, 220
coupling, object, 483
Courington, William, 799
Covington, Michael, 324
Cramer, Bill, 289
"Creating an Adventurous Language," 173
CrossCode C, 449
crotchets, 433, 513, 562, 629, 682
cryptography, 412
crypto-algorithms, 412
C-scape On-Line Reference, 110
CSP, 271
C_talk browser, 677
C_talk, 677, 389
CXDB, 52
C-Worthy, 521
Cypress Systems, 703

Dahlgren, Kent, 592
DAIR Computer Systems, 519
DAMacMan, structure of, 15,
installing, 16
data compaction, 412
data declaration, 789
data format, 600
data integrity, 125
data operations, 32-bit, 539
data structures, 25, 600
database applications, 600
Datalight Optimum-C, Version 3.12, 402
DataPlex, 451
data, stripped, 236
data-entry screens, 561
dBASE Reflex, 263
dBFind, 110
DE, 274
debuggers, 718
debugging, 419, 630
hardware support for, 61
"Debugging with the 80386," 61
decompiler, 301
decomposing programs, 462
Demo II, 279
Derossi, Chris, 664
design documentation, 408
desk accessory, 10
writing, 13
desktop publishing, 234
desktop, Actor, 43
developer, Macintosh, 714
Developer Relations program, 279
Developer's Toolkit for C Language, The, 389
"Developing for the User," 231
diagramming, action-chart, 459
DigiBoard, 164
Digitalk, 221, 610
Dijkstra, E.W., 119
Dilworth, John B., 533
DIR command, 440
disk-based array, 557
disk error, 96
Disk Software, 51
Dis*Doc, 574
documentation, 408
DOS critical error handler, writing, 96
Dos Reis, Anthony J., 185
DOS/16M, Version 2.0, 640
"Double Cross for MASM, A," 541
Drang, Diane E., 221
drivers, 751
DSI Signal-to-Disk Interface, 576
d-tree, 388
Duncan, Ray, 659
Duplin, Doreen, 33
Duvanenko, Victor J., 597
Dyalog APL, 333
dynamic linking, 172
"Dynamic Run-time Structures," 600

Echo, 110
Edelson, Barry, 221
"Editorial," 3, 57, 115, 169, 227, 285, 337, 393, 455, 525, 581, 645, 762, 767, 825
editors, 405
EGA, fonts, 128
"EGA and VGA: Smooth Scrolling and Panning," 671
elements, 600
Elsevier Science Publishing Company, 521
environmental hierarchies, 801
EPEC, 388
Erickson, Jonathan, 227, 285, 337, 393, 455, 525, 577, 581, 642, 646
error, hardware,
event flags, 291
event-driven programs, 712
EVERCOM II 24, 164
Everex, 164
"Examining Room," 103, 158, 217, 273, 327
Execution Executive, 790
Expanded Entertainment, 163-164
ExperTelligence, 51

FaceIt, 451, 641

FariCom, 388
Fastmath, 388
Feiber, Jonathan, 799
file system, 801
file-management, 680
Fillmore Systems, 110
Finder, Macintosh, 595
"Find That Function!," 404
"Find That Function—From Inside Brief!" 674
flags, 664
Flambeaux Software, 281
floating point, 315
"Flotsam and Jetsam," 201, 257
Floyd, Robert, 266, 633
fonts, EGA, 128
FORCE, 99
FORML, 99, 214
Forth, 533
Forth, in AI, 100
Forth, structure, 535
Forth Conference, Rochester, 314
Forth Covention, 314
FORTH Inc., 99, 575
Forth Modification Laboratory, 99
Forth Modification Laboratory Conference, Ninth, 214
"Forth Column, The," 564
Fortrix-C, 332
Freeman, Neil, 329
Frey, Donnalyn, 655
Froth Interest Group, 564
FTL, 485
FTL Modula-2, 52, 332
functions, 404, 405
  memory management, 604
fundamental units, 481

Gasper, Elon, 341
Geary, Michael, 349
General Programming Resource Guide, 733
Genesis 1024, 279
Genus, 640
Genus Microprogramming, 388
gestalt, approach to pattern matching, 355
getf.bldfuncs, 404
"Getting A Head with HyperAnimation," 341
"Getting Down to BASICs," 359
GHC Tools, 640
Gimpel Software, 110
glossary, for Software Engineering Sourcebook, 817
Gold Hill Computers, 220
Golden Common Lisp Developer 286, 220
Golden Solution, 273
Graph X, Version 3.0, 640
graphical interfaces, 592
graphical programming, 787
Graphic Device Interface, 640
Graphic Software Systems, 281
graphics display resolutions, 597
graphics engine, 593
Grosberg, John A., 478
Grossman, Franklin, 61
GSS Graphics Development Toolkit for OS/2, 281
Guidelines C++, 219

Hamilton Group, 655
"Handling Image Files with TIFF," 234
Harbison, Samuel P., 95
hardware error, 96, disk, nondisk, 96
Harris Semiconductor, 99, 564
Harvard architecture machine, 9
head, talking, 341
Hecht-Nielsen Neurocomputer, 705
Helios Software, 641
Heller, Steve, 541
Hercules Computer Technology, 640
Herzfeld, Andy, 634
Hewlett-Packard, 450
HFS Navigator, 206
high-level languages, 131
Hildebrand, Dan, 294
HiSoft, 332

Holub, Allen, 24, 91, 117, 200, 255, 311, 376
Homebrew Computer Club, The, 326
Honda, Masahiro, 799
HPLaserROM, 450
huge array matrices, 557
huge arrays, 155
Hugemats, 287
Human Intellect Systems, 51
Human Interface Guidelines, 149, 262, 715
hung programs, 62
Hymowech, Marvin, 404, 674
HyperAnimation, 341
HyperAnimator, 347
HyperBase, 705
HyperCard, 4, 148, 206, 341, 497, 720, 749
HyperCard Developer's Toolkit, 260
HyperCard Resource Guide, 752
HyperCard Script Language Guide, 260
HyperCard Stack Design Guidelines, 260
HyperCard Technical Reference, 260
hyperICE-386, 575
HyperTalk, 206, 751
HyperText, 183, 342, 633

IBM, 450
Ibuki, 574
Iceview, 641
IEEE International Conference on Neural Networks, 566
IGC, 333
image compilation, 597
"Image Compression Via Compilation," 597
image file directory, 236
image file header, 236
images, compressing, 236
IMS BO14, 705
incremental implementation, 510
INMOS International, 705
"Inserting Elements into a Basic Integer Array," 608
Inside Macintosh, 10, 715
Inside Macintosh X-Ref, 715
Intel, 7
Intel 80386, 61
Intel 8088, 8
Intel Corporation, 641
interactive, 289
internal text formats, 626
intertask communications, 294
inter-tool, 806
intra-tool, 806
"Introduction to HyperCard Programming, An," 749
IOPL, 661
Isogon Corp., 164
"Italic Font in C for the EGA and VGA, An," 128

Jandi Technologies, 388
Jensen and Partners International, 487

Kahl, Michael, 33
Kalinsky, David, 793
kernel, multitasking, 65
Kernighan, Brian W., 279, Brian, 378, Brian, 521
keyboard macros, 459
Kilpatrick, James J., 433
Kinetic Access, 164
Kinetic Corp., 164
King, K.N., 466
King, Todd, 600
Knaster, Scott, 259, 433
Knowledge Garden, 640
Knowledge Navigator, 341
KnowledgePro, 181, 640
Koopman, Phil, 99
Krute, Stan, 31, 148, 205, 259, 497
Kuhn, Thomas S., 383

Laboratory Microsystems, 332
LabVIEW, 790
Ladd, Scott Robert, 668
Language Extensions I, 704
languages, high-level, 131
languages, programming, 715

languages, text, 787
large objects, 126
Laser DataBank, 52
Lattice Inc., 450, 521
Lattice C Compiler, 521
Lattice dBC, 24
  library, 28
Lazarev, Gregory L., 610
Leblang, David B., 803
"Letters," 59, 117, 171, 229, 287, 339, 394, 456, 527, 583, 647
Levine, Robert, 221
Lewis, Ted G., 10, 778
Lightspeed, C, 33
Lim 4.0, 320
Lindley, Craig A., 65
LineDrawing, 210
Linett, Charles, 265
Lingwood, Dave, 31
linked list, 133
Lisa, 716
LISP, 176, 529
  records in, 532
Lisp Wizard, 574
list, linked, 133
list processing, 184
Livshin, David, 274
LMI Forth Metacompiler, 332
logic programming, 175, 610
Logic Programming Associates, 51
Logitech, 486
LPA MacPROLOG, 51
Lulay Backdoor, 703
Lulay Software, 703
Lynx Real-Time Systems, 450
LynxOS, 450

Mac AI, 51
MacApp, 730
Mac applications, structure of, 11
Mac code, 33
Mac DA, structure of, 13
MacDraw, 720
MacForth, 438
Macintosh, 7, 10, 664
Macintosh developer, 714
Macintosh development, internal, 720
Macintosh Finder, 595
Macintosh operating system, 712
Macintosh Programming Secrets, 259, 433
Macintosh Revealed, 715
Macintosh, upgrade, 32
MacMan, 10, using, 10
  tools, 15
Mac OS, 664
"Mac Programmer's Resource Kit, A" 712
macros, 461
MacSEF, 714
MacSRMS, 451
Macworld Expo, 259, 634
MAKE files, 681
"Mapping DOS Memory Allocation," 603
Margulis, Neal, 537
marktime(), 292
MASM, 541
Mathews, James, 133
matrices, 156
  huge array, 557
May, William, 301
MC68020, 388
MCB, 603
McCPrint, 450
McGraw-Hill, 221
Meadow, Anthony, 234, 712, 741
MEMOCOM, 52
memory, 8
memory allocation, 603
memory control block, 603
memory management functions, 604
memory segmentation, 155
MEMULATOR, 52
menu, window, 560
menu bars, 379
menus, as objects, 41
menu structure, 680
message passing, 295, 300
"Message-Passing Operating Systems," 294
messages, 120, 478

Metacomco, 51
Toolkit, 51
metafiles, 594
Metaphor, 703
Metzener, David E., 355
MHT Software, 111
Michener, John R., 412
Micro Computer Control, 389
Micro Focus, 704
Microcosm Inc., 575
MicroProducts, 220
Microsoft, 106, 161, 169, 317, 574, 659, 703, 704
Microsoft C, 117
5.1, 399, 397
Microsoft C-compatible library, 671
Microsoft Macro Assembler 5.0, 106
Microsoft MASM, 492
Microsoft MB 6.0, 362
Microsoft Mixed-Language Macros, 496
Microsoft QuickBasic 4.0, 608
Migent, 389
Miller Freeman's Software Development '88, 322
MIMD parallelism, 267
MIMD programming, 270
Minsky, Marvin, 566
MIPS Software Development, 333
MKS Toolkit, 575
MMC AD Systems, 450
modular, 650
modularity, 517
Modula-2, 210, 485, 517
Modula-2/68, 163
Moon, Raymond, 492
Moore, Robert J., 603
Mortice Kern Systems, 575
Mountain View Press, 99
mouse, 11
MPW interface, 720, 722, 737
background, 738
MS-DOS, 668
Version 2.0, 603
MS-Pascal 4.0, 317
MultiAPL, 705
MultiFinder, 148, 664
tricks, 666
multiple inheritance, 322
multitasking kernel, 65
preemptive, 24

nap(), 292
National Design Inc., 279
Nelson, Ted, 282
nested topics, 183
network computing technology, 804
Netwurkz, 519
neural nets, 386
neural network paradigm, 519
New Basics, 360
NEWSPACE, 164
nondisk error, 96
Northwest Computer Algorithms, 52
Norton, Peter, 103
Norton Guides, The, 103, 327
notation, postfix, 255, reverse-Polish notation, 255
notification manager, 667
Novix 4000, 99
Nute, Donald, 324
NuTools, 703
Nutter, Stewart, 408
Nu-Mega Technologies, 275

Oakland Group, 110
Oasys, 521
OASYS 80386, 333
object coupling, 483
object oriented programming, 322
Objective-C, 163
ObjectLogo, 163
objects, 120
objects, large, 126
object-oriented C, 323
object-oriented data management system, 124
object-oriented database systems, 322
"Object-Oriented Dimensional Units," 478
Object-Oriented Forth, 215
object-oriented hybrid, 677

object-oriented model, 120
object-oriented Pascal, 39
object-oriented programming, 176, 272, 610, 729
"Object-Oriented Programming and Databases," 119
"Objective C," 416
Occam, 271, 325
Offner, Rocky, 234
"Of Interest," 51, 110, 163, 220, 279, 332, 388, 449, 521, 574, 640, 703

Ohm's law, 478
OOD, 478
Open Software Foundation, 655
OpenEnder, 164
operating systems, 294, 650
  Macintosh, 712
  organization, 24
Operating Systems: Design and Implementation, 24
Opotech, 52
OPS83, 221
optimization, 398
Optimum C, Datalight, 402
Opus Systems, 574
OP-TASM, 574
Oregon C ++ Software Development System, 449
Oregon Software, 449
OSF, 655
OS/2 Unix Multitasking Benchmark, 450
OS/2, 297, 659
overlays, 631
overloading, 468
"Overview: Macintosh Programmer's Workshop," 720
O'Neill Software, 111

panning, smooth, 671, 672
Paperback Software, 221
paradigms, 686
  analysis, 688
  clash, 383
parallel, sorting, 443
parallel algorithms, 443, 686
parallel architectures, 687
Parallel processing, 325
parallel programming, 339
parallelism, 323
Parker, Jeffrey M., 786
parsing, 683
Pascal, 263, 287, 359, 514
  4.0, 317
Pascal, vs. Ada, 466
passed arguments, 492
"Pattern Matching: The Gestalt Approach," 355
PC CASE, 804
PCX Programmer's Toolkit, 388, 640
PC/Forms, 273
Peabody for Turbo C, 327
performance requirements, 510
Peter Norton Computing, 110
"Photorealism in Computer Graphics," 585
PL/D, 519
pop up, 379
portability, 650
Porter, Kent, 155, 161, 210, 218, 219, 263, 274, 317, 328, 331, 379, 440, 485, 514, 557, 630, 683, 96
POSIX, 656
postfix notation, 255
Postscript, 594
Pountain, Dick, 215
PowerBasic, 332
PowerLisp version 2.0, 220
Prentice Hall, 279, 521
print functions, formatted, 200
printf(), 200
priority-queue routines, 28
Production Systems Technologies, 221
Productivity Products International, 163
Professional MultiTasker, The, 333
Professional Version 1.2, 332
program segment prefix, 668
program structure, 478

programmers, C, 242
Programmer's Library, 574
Programmer's Online Companion, 52
"Programmers' Database for the Macintosh Toolbox, A," 10
programming contest, 438
Programming in Modula-2, 210
programming languages, 715
programming system, 786
programming, experimental and conceptual, 773
programming, graphical, 787
programming, large-scale efforts, 799
programming, logic, 175, 610
object oriented, 176, 610
programming, parallel, 339
"Programming Paradigms," 266, 322, 383, 443, 519, 566, 633, 686
Programs in Motion, 280
programs, decomposing, 462
programs, event-driven, 712
programs, hung, 62
programs, resident, 131
Prolog, 324
terminology, 615
Prolog Programming in Depth, 324
"Prolog/V: Prolog in the Smalltalk Environment," 610
protected mode, 537, 659
Proteus, 641
prototyping, theory of, 781
pull down, 379
"Putting Graphical Interfaces into Perspective," 592

Quantum Software, 575
QUED/M 2.04, 205
Quick Basic 3.0, 360
Quick Basic 4.0, 361
Quick C, 117
Quilt Computing, 451
Quinn-Curtis, 164

Rapitech Systems, 332
RasterOps Corp., 164
Ratcliff, John W., 355
Ratcliff/Obershelp pattern matching algorithm, 355
Rational Systems, 640
RAVE, 342, 343, database structures, 346
RAVEL, 342, 343
Ready, Jim, 793
Ready Systems, 574
read-write, 122
real time, 289, 457
Real Time Express, 437
real-time environments, 294
real-time operating system, 314
Real-Time Programming Conference, 437
Real-Time Programming Convention, 564
"Rebirth of the Macintosh," 711
Red Ryder, 634
Reduce, 52
Reinhardt, Dennis, 519
Relph, Richard A., 162, 219, 276, 397
Renaissance, 640
Renaissance Graphics Device Interface Developer's Kit, 279
Renaissance GRX, 279
RenderMan specification 591, 595
requirements analysis, 510
resident programs, 131
resolution, completeness, 187,
semantic, 187
resolution principle, 185, 186
resolutions, for graphics displays, 597
Resource Guide, 807
reverse-Polish notation, 255
RFT, 656
RGDI, 279
Ritchie, Dennis M., 279, 378, 521
RJ Swantek & Associates, 574
Robinson, J.A., 185
Robinson, Phillip, 767
Rochester Forth Conference, 564
ROM routines, 10
rotor cryptographic operator, 412

RS-232 serial protocol analyzer, 65
RTX 2000
Ruddy, Sara Noah, 3
run-length encoding, 599
"Running Light," 4, 58, 116, 170, 228, 286, 338, 384, 456, 526, 582
Rvalues, 257

scan lines instructions, 599
scheduler, 290
Schindler, Max, 314
Scouting Toolkit, 208, 261
screen control, 560
scrolling, smooth, 671, 672
semantic clashes, 189
send/receive message, 298
Serengeti Software, 111
"Serial Protocol Analyzer Program, A," 65
setf, 531
shading language, 587
Shammas, Namir Clement, 39
Shapiro, Alen D., 221
shell archives, 376
Show Me!, 111
SIGGRAPH, 585
"Simple Decompiler, A," 301
Singer, Andrew, 33
SLR Systems, 574
Smalltalk, 774
Smalltalk, terminology, 613
Smalltalk/V, 610
Small, Charles, H., 314
smooth panning, 671, 672
smooth scrolling, 671, 672
SoftCode, 574
Soft Evolutions, 640
SoftRex, 110
SoftScience Corp, 280
software accelerators, 784
"Software and the Single Programmer," 778
Software Bottling Company, The, 574
software copyright, 57
software development, 231
software development effort, large, 803
Software Development Factory, 110
Software Development Kit, 703
Software Development Systems, 111, 449
"Software Engineering Environments," 770
Software Garden, 279
Software Link, The, 279
Soft-ICE, 275
Soft-Scope Source-Level Debugger, 104
Solution Systems, 218, 521
Source Print, 459
SPA, 65
SPA, connecting, 66
  using, 67
  program's architecture, 69
  menuing system, 70
  enhancements, 71
"Speed Trials: Five Cs Compared," 397
spelling, checker, 133
"Spelunking MS-DOS: Documenting the Undocumented," 668
Spencer Organization, 705
Sperry, Tyler, 7, 57, 58, 115, 116, 169, 170, 228, 286, 338, 394
spreadsheets, software, 774
SQL development kits, embedded, 352
"SQL Development Tools," 349
SQLWindows, 349
"State-of-the-Art in Modula-2, The," 485
Steele, Guy L. Jr., 95
Stein, Jacob, 119
Stepstone, 416
Sterling Castle, 110
Stevens, Al, 432, 510, 560, 626, 680
Stitt, Martin, 459
Stony Brook, 488
Stony Brook Software, 210
strings, 263
structure, 600
structure, of program, 478

Structure of Scientific Revolutions, The, 383
Structured Induction in Expert Systems, 221
"Structured Programming," 39, 96, 155, 210, 263, 317, 379, 438, 514, 557, 630, 683
structured query language, 349
STSC, 705
subroutine, 27
Sun Microsystems, 655
Sun-3, 449
SuperMac Technology, 262
support functions, 303
Sutherland, Randy, 711, 112, 165, 223
Swaine, Michael, 4, 53, 266, 282, 322, 334, 383, 390, 443, 452, 456, 519, 522, 566, 633, 686, 706, 762, 825
"Swaine's Flames," 53, 112, 165, 223, 282, 334, 390, 452, 522, 577, 642 706
Symbolics, 222
symbols, 345
synchronous system requests, 290
system, extending, 529
system administrator tasks, 296
System Sleuth, 575
System Software Tools, 24

table file, 263
"Tackling Large-Scale Programming Projects," 799
Tag Image File Format, 234
tags, 236
talking head computer interface, 341
Talking Tiles, 342
Tanenbaum, Andrew, 24
TASM, 630
taxonomy, 686
TDS, 325
TECH Help! Version 3.3A, 281
Tello, Ernest R., 43, 416, 545, 677
Texas Instruments, 703
Text Collector, 111
text editor, window, 626
text languages, 787
text searching functions, 680
"Theorem Proving Using Semantic Resolution," 185
THEOS 386, 703
THEOS Software, 703-704
Think Technologies, 33
Thompson, Bill and Bev, 181
"Threaded Binary Trees," 133
Tichnor, Mark, 242
TIFF, 234, structure, 235
TIFF, problems with, 238, sample program, 240
tiny word processor, 680
TMON 2.81, 205
tokens, 683
Tonkin, Bruce, 359, 608
Toolbox, 10
"Tool for Secret Key Cryptography, A," 412
Tools & Techniques, 451
toolsets, 793
tools, 232
"To The Macs," 31, 148, 205, 259
"Topics in Knowledge-Based Languages," 181
TopSpeed Modula-2,l 487
Tracy, Martin, 99, 214, 314, 437, 564
transputer, 270, 325
tree structure, 408
Tree86 Version 1.1, 333
Trilogy, 221, 451
True BASIC, 51
True BASIC Inc., 51
TSR function, 669
Turbo Apprentice 5.0, 703
Turbo Assembler, 630
Turbo Basic 1.1, 360
Turbo C, 117, 379
Turbo C, Borland's, 401
Turbo C 2.0, 630
Turbo C Run-time Library Source, 333
Turbo Pascal, 379

Turbo Pascal 4.0, 96, 158, 263, 265, 440
Turbo Pascal 5.0, 317, 630
Turbo Power Software, 217, 330
Turbo Professional 4.0, 217
Turbo Prolog 2.0, 545
Turbo Shell Version 2.0, 221
TurboGeometry Library, 51
Turing Award, 633
Turing Award Lecture, 266
TWRP, 680
T-DebugPLUS 4.0, 330

uninitialized pointer, 311
units, fundamental, 481, composite, 481
UniWare, 111
UNIX, 659, 742, 772
  shells, 376
"Unix vs. Unix," 655
Upstill, Steve, 585
USENIX, 657
user interface, 592
"Using Action Charts for Software Development," 459
"Using an API as a Developer Platform," 786

V Communications, 280
validation, ada, 474
variables, automatic, 492, 493
Vellino, Andre, 324
Vici interpreter, 419
virtual array management scheme, 242
virtual arrays, using, 245, implementing, 245
"Virtual Arrays in C," 242
virtual-machine capability, 61
visual programming, 780
VM Personal Computing, 333
VP-Expert, 221
VRTX32, 574
VRTX32, 574

V.I. Corp., 703

WATCOM, 163
WATCOM, Express C, 163
Watcom C, 401
  Version 6.5,397
WATCOM C 6.0, 163
Watson, Scott, 634
West Coast Computer Faire, 1984, 259
Whitesmiths, 52
Whitesmiths Ltd., 389
Whitewater Group, The, 43, 704
WildCard, 749
WildTalk, 749
window library, 510
Window Management System, 111
window manager, 592
window menu, 560
windows, 596
Winograd, Terry, 770
Wirth, Nicklaus, 210, 514
word processor, tiny, 680
Workman Associates, 52, 485
World Wide Data, 279
Writer's Art, The, 433
"Writing OS/2 Applications with I/O Privileges," 659
"Writing Programs for MultiFinder," 664
"Writing Real-Time Program Under Unix," 289
WYSIWYG, 671
Wyte Corp, 105

XCMD, 751
"XCMD and XFCN: HyperCard's Software Slots," 497
XFCN, 751
XO-Shell, 105
XREF, 683
Xxref, 541

More Programming Tools from M&T Books

Programming Languages

C

Graphics Programming in C
Roger T. Stevens
Item #019-2 $39.95 (book/disk)
Item #018-4 $24.95 (book)
Details the fundamentals of graphics processes for the IBM PC family and its clones. All the information needed to program graphics in C, including source code, is presented. The provided source code will enable the user to easily modify graphics functions to suit specific needs. Both Turbo C and Microsoft C are supported.

C Chest and Other C Treasures from *Dr. Dobb's Journal*
Edited by Allen Holub
Item #40-2 $24.95 (book)
Item #49-6 $39.95 (book/disk)
This comprehensive anthology contains the popular "C Chest" columns from *Dr. Dobb's Journal of Software Tools*, along with the lively philosophical and practical discussions they inspired, in addition to other information-packed articles by C experts. The software in the book is also available on disk with full source code. MS-DOS format.

Turbo C: The Art of Advanced Program Design, Optimization, and Debugging
Stephen R. Davis
Item #38-0 $24.95 (book)
Item #45-3 $39.95 (book/disk)
Overflowing with example programs, this book fully describes the techniques necessary to skillfully program, optimize, and debug in Turbo C. All programs are also available on disk with full source code. MS-DOS format.

A Small C Compiler: Language, Usage, Theory, and Design
James E. Hendrix
Item #88-7 $23.95 (book)
Item #97-6 $38.95 (book/disk)
A full presentation of the design and theory of the Small C compiler (including source code) and programming language. The author has implemented many features in this compiler that make it an excellent example for learning basic compiler theory. Some of these features are: recursive descent parsing, one-pass compilation, and the generation of assembly language. Here is a look into a real compiler with the opportunity for hands-on experience in designing one.

Dr. Dobb's Toolbook of C
Editors of *Dr. Dobb's Journal*
Item #89303-615-3 $29.95
From *Dr. Dobb's Journal of Software Tools* and Brady Communications, this book contains a comprehensive library of valuable C code. *Dr. Dobb's Journal of Software Tools'* most popular articles on C are updated and reprinted here, along with new C programming tools. Also included is a complete C compiler, an assembler, text processing programs, and more!

The Small-C Handbook
James E. Hendrix
Item #8359-7012-4 $17.95 (book)
Item #67-4 $37.90 (book and CP/M disk)
Also from *Dr. Dobb's Journal of Software Tools* and Brady Communications, the handbook is a valuable companion to the Small-C compiler, described below. The book explains the language and the compiler, and contains entire source listings of the compiler and its library of arithmetic and logical routines.

Forth

Dr. Dobb's Toolbook of Forth
Edited by Marlin Ouverson
Item #10-0 $22.95 (book)
Item #57-7 $39.95 (book/disk)
This comprehensive collection of useful Forth programs and tutorials contains expanded versions of *Dr. Dobb's Journal of Software Tools*' best Forth articles and other material, including practical code and in-depth discussions of advanced Forth topics. The screens in the book are also available on disk as ASCII files in the following formats: MS/PC-DOS, Apple II, Macintosh, or CP/M: Osborne or 8" SS/SD.

Dr. Dobb's Toolbook of Forth, Volume II
Editors of *Dr. Dobb's Journal*
Item #41-0 $29.95 (book)
Item #51-8 $45.95 (book/disk)
This complete anthology of Forth programming techniques and developments picks up where the Toolbook of Forth, First Edition left off. Included are the best articles on Forth from *Dr. Dobb's*

Journal of Software Tools, along with the latest material from other Forth experts. The screens in the book are available on disk as ASCII files in the following formats: MS-DOS, Macintosh, and CP/M: Osborne or 8" SS/SD.

BASIC

QuickBASIC: Programming Techniques and Library Development
Namir Clement Shammas
Item #004-4 $34.95 (book/disk)
Item #003-6 $19.95 (book)
This book provides the reader with the opportunity to learn the details of creating subroutines, functions, and libraries to permit more structured coding.The remainder of the book is dedicated to an in-depth discussion of building original libraries and functions to fulfill individual programming needs. Programs and subroutines are available on disk with full source code.

Turbo BASIC: Programming Techniques and Library Development
Namir Clement Shammas
Item #016-8 $34.95 (book/disk)
Item #015-X $19.95 (book)
Advanced programmers will be introduced to the flexible Turbo BASIC environment, programming framework, data types, and the use of libraries, functions and subroutines to permit more structured coding. As with the QuickBASIC book, the techniques discussed in this volume are then put to use building a selection of useful libraries. All programs and subroutines are also available on disk with full source code.

HyperTalk

Dr. Dobb's Essential HyperTalk Handbook
Michael Swaine
Item #99-5 $39.95 (book/disk)
Item #99-0 $24.95 (book)
Well-known columnist Michael Swaine provides a complete analyses of HyperTalk in this new book. Complete coverage of topics such as the move from authoring to scripting, concepts and components of the language, programming style considerations, full language exposition and discussion, and more, are presented. Programs available on disk.

MIDI

C Programming for MIDI
Jim Conger
Item #86-0 $22.95 (book)
Item #90-9 $37.95 (book/disk)
For musicians and programmers alike, here is the source that will help you write programs for music applications. The author begins by outlining the features of MIDI (Musical Instrument Digital Interface) and its support of real-time access to musical devices. An introduction to C programming

fundamentals as they relate to MIDI is also provided. The author fully demonstrates these concepts with two MIDI applications: a patch librarian and a simple sequencer.

MIDI Programming for the Macintosh
Steve De Furia and Joe Scacciaferro, Ferro Technologies
Item #022-2 $37.95 (book/disk)
Item #021-4 $22.95 (book)
This book equips the musician and programmer alike with the background necessary to program music applications and to take advantage of all the Macintosh and the MIDI interface have to offer. Specific examples are presented and all source code is available on disk.

Business

PC Accounting Solutions
Editors of *PC Accounting* (formerly *Business Software*)
Item #008-7 $37.95 (book/disk)
Item #009-5 $22.95 (book)
This anthology serves as a well-rounded source of expert information for managers who want to implement a PC-based accounting system or to gain better control of their existing system. From choosing and maximizing your accounting systems and software to building better spreadsheets and budgets, this book is an immensely valuable source that will improve your ability to analyze the information that is critical to the success of your business.

Public-Domain Software and Shareware: Untapped Resources for the PC User, Second Edition
Rusel DeMaria and George R. Fontaine
Item #011-7 $19.95 (book)
Item #014-1 $34.95 (book/disk)
Organized into a comprehensive reference, this book introduces the novice and guides the experienced user to a source of often overlooked software—public domain and Shareware. This book will tell you where it is, how to get it, what to look for, and why it's for you. The sample programs and some of the software reviewed is available on disk in MS-DOS format. Includes information on how to obtain free access time on CompuServe, and other special offers.

Time and Task Management with dBASE III
Timothy Berry
Item #09-7 $49.95 (manual/MS-DOS disk)
Like an accounting system for time and tasks, this package helps users organize hours, budgets, activities, and resources. Providing both a useful time-management system and a library of dBASE III code and macros, this package has practical as well as educational value. To be used with dBASE III. Source code and documentation is included. MS-DOS disk format.

Sales Management with dBASE III
Timothy Berry
Item #15-1 $49.95 (manual/MS-DOS disk)
Sales management works with dBASE III to provide a powerful information system that will help you to keep track of clients, names, addresses, follow-ups, pending dates, and account data. This system organizes all the day-to-day activities of selling and includes program files, format files, report files, index files, and data bases. Documentation and full source code is included.

Programming Tools and Source Code Libraries

C

Small-Windows: A Library of Windowing Functions for the C Language
James E. Hendrix
Item #35-X $29.95
Small-Windows is a complete windowing library for C. The package includes video functions, menu functions, window functions, and more. The package is available for MS-DOS systems for the following compilers: Microsoft C Version 4.0 and 5.0; Small-C; Turbo C 1.0 and 1.5; and Lattice C 3.1. Documentation and full C source code is included.

Tools

Small Tools: Programs for Text Processing
James E. Hendrix
Item #78-X $29.95 (manual/disk)
This package of text-processing programs written in Small-C is designed to perform specific, modular functions on text files. Source code is included. Small Tools is available in both CP/M and MS/PC-DOS versions and includes complete documentation.

Small Assembler: A Macro Assembler Written in Small C
James E. Hendrix
MS-DOS version: Item #024-9 $29.95 (manual/disk)
CP/M version: Item #77-1 $29.95 (manual/disk)
Here is a full macro assembler which was developed primarily for use with the Small-C compiler. It provides an excellent example for learning the basics of how assembler works. The manual provides an overview of the Small Assembler, documents the command lines that invoke programs, and more. The accompanying disk includes both the executable assembler and full source code.

NR: An Implementation of the UNIX NROFF Word Processor
Allen Holub
Item #33-X $29.95
NR is a text formatter that is written in C and compatible with UNIX's NROFF. *NR* comes configured for any Diablo-compatible printer, as well as Hewlett-Packard's ThinkJet and LaserJet.

Both the ready-to-use program and full source code are included. For PC compatibles.

Turbo Pascal

Statistical Toolbox for Turbo Pascal
Namir Clement Shammas
Item #22-4 $39.95 (manuals/disks)
Two statistical packages in one! A library disk and reference manual that includes statistical distribution functions, random number generation, basic descriptive statistics, parametric and nonparametric statistical testing, bivariate linear regression, and multiple and polynomial regression. The demonstration disk and manual incorporate these library routines into a fully functioning statistical program. For IBM PCs and compatibles.

Turbo Advantage
Lauer and Wallwitz
Item #26-7 $29.95
A library of more than 200 routines, with source code sample programs and documentation. Routines are organized and documented under the following categories: bit manipulation, file management, MS-DOS support, string operations, arithmetic calculations, data compression, differential equations, Fourier analysis and synthesis, and much more! For MS/PC-DOS systems.

Turbo Advantage: Complex
Lauer and Wallwitz
Item #27-5 $39.95
This library provides the Turbo Pascal code for digital filters, boundary-value solutions, vector and matrix calculations with complex integers and variables, Fourier transforms, and calculations of convolution and correlation functions. Some of the *Turbo Advantage: Complex* routines are most effectively used with Turbo Advantage. Source code and documentation included.

Turbo Advantage: Display
Lauer and Wallwitz
Item #28-3 $39.95
Turbo Advantage: Display includes an easy-to-use form processor and thirty Turbo Pascal procedures and functions to facilitate linking created forms to your program. Full source code and documentation are included. Some of the *Turbo Advantage* routines are necessary to compile *Turbo Advantage: Display*.

Operating Systems

OS/2

The Programmer's Essential OS/2 Handbook
David E. Cortesi
Item #82-8 $24.95 (book)
Item #89-5 $39.95 (book/disk)
Here is a resource no developer can afford to be without! Cortesi succinctly organizes the many features of OS/2 into related topics and illuminates their uses. Detailed indexes and a web of cross referencing provide easy access to all OS/2 topic areas. Equal support for Pascal and C programmers is provided. *The* essential reference for programmers developing in the OS/2 environment.

UNIX

UNIX Programming on the 80286/80386
Alan Deikman
Item #83-6 $24.95 (book)
Item #91-9 $39.95 (book/disk)
A complete professional-level tutorial and reference for programming UNIX and XENIX on 80286/80386-based computers. Succinct coverage of the UNIX program environment, UNIX file system, shells, utilities, and C programming under UNIX are covered. The author also delves into the development of device drivers; some examples of these are video displays, tape cartridges, terminals, and networks.

On Command: Writing a UNIX-Like Shell for MS-DOS
Allen Holub
Item #29-1 $39.95
Learn how to write shells applicable to MS-DOS, as well as to most other programming environments. This book and disk include a full description of a UNIX-like shell, complete C source code, a thorough discussion of low-level DOS interfacing, and significant examples of C programming at the system level. All source code is included on disk.

/util: A UNIX-Like Utility Package for MS-DOS
Allen Holub
Item #12-7 $29.95
This collection of utilities is intended to be accessed through SH but can be used separately. It contains programs and subroutines that, when coupled with SH, create a fully functional UNIX-like environment. The package includes a disk with full C source code and documentation in a UNIX-style manual.

MS-DOS

Taming MS-DOS, Second Edition
Thom Hogan
Item #87-9 $19.95
Item #92-5 $34.95
Described by reviewers as "small in size, large on content," and "fun." The second edition promises to be just as readable and is updated to cover MS-DOS 3.3. Some of the more perplexing elements of MS-DOS are succinctly described here with time-saving tricks to help customize any MS-DOS system. Each trick is easily implemented into your existing tools and for programmers, Hogan includes many complete source code files that provide very useful utilities. All source code is written in BASIC.

Program Interfacing to MS-DOS
William G. Wong
Item #34-8 $29.95
Program Interfacing to MS-DOS will orient any experienced programmer to the MS-DOS environment. The package includes a ten-part manual with sample program files and a detailed description of how to build device drivers, along with the device driver for a memory disk and a character device driver on disk with macro assembly source code.

Other

Tele Operating System Toolkit
Ken Berry
This task-scheduling algorithm drives the Tele Operating System and is composed of several components. When integrated, they form an independent operating system for any 8086-based machine. Tele has also been designed for compatibility with MS-DOS, UNIX, and the MOSI standard.

SK: THE SYSTEM KERNEL
Item #30-5 $49.95 (manual/disk)
The System Kernel contains an initialization module, general-purpose utility functions, and a real-time task management system. The kernel provides MS-DOS applications with multitasking capabilities. The System Kernel is required by all other components. All source code is included on disk in MS-DOS format.

DS: WINDOW DISPLAY
Item #32-1 $39.95 (manual/disk)
This component contains BIOS level drivers for a memory-mapped display, window management support and communication coordination between the operator and tasks in a multitasking environment. All source code is included on disk in MS-DOS format.

FS: THE FILE SYSTEM
Item #65-8 $39.95 (manual/disk)
The File System supports MS-DOS disk file structures and serial communication channels. All source code is included on disk in MS-DOS format.

XS: THE INDEX SYSTEM
Item #66-6 $39.95 (manual/disk)
The Index System implements a tree-structured free-form database. All source code is included on disk in MS-DOS format.

Chips

Dr. Dobb's Toolbook of 80286/80386 Programming
Edited by Phillip Robinson
Item #42-9 $24.95 (book)
Item #53-4 $39.95 (book/disk)
This toolbook is a comprehensive discussion of the powerful 80X86 family of microprocessors. Editor Phillip Robinson has gathered the best articles from numerous key programming publications to create this valuable resource for all 80X86 programmers. All programs are available on disk with full source code.

Dr. Dobb's Z80 Toolbook
David E. Cortesi
Item #07-0 $25.00 (book)
Item #55-0 $40.00 (book/disk)
This book contains everything users need to write their own Z80 assembly-language programs, including a method of designing programs and coding them in assembly language and a complete, integrated toolkit of subroutines. All the software in the book is available on disk in the following formats: 8" SS/SD, Apple, Osborne, or Kaypro.

Dr. Dobb's Toolbook of 68000 Programming
Editors of *Dr. Dobb's Journal*
Item #13-216649-6 $29.95 (book)
Item #75-5 $49.95 (book/disk)
From *Dr. Dobb's Journal of Software Tools* and Brady Communications, this collection of practical programming tips and tools for the 68000 family contains the best 68000 articles reprinted from *Dr. Dobb's Journal of Software Tools,* along with much new material. The book contains many useful applications and examples. The software in the book is also available on disk in the following formats: MS/PC-DOS, Macintosh, CP/M 8", Osborne, Amiga, and Atari 520ST.

X68000 Cross Assembler
Brian R. Anderson
Item #71-2 $25.00
This manual and disk contain an executable version of the 68000 Cross Assembler discussed in *Dr. Dobb's Toolbook of 68000 Programming,* complete with source code and documentation. The

Cross-Assembler requires CP/M 2.2 with 64K or MS-DOS with 128K. The disk is available in the following formats: MS-DOS, 8" SS/SD, and Osborne.

General Interest

DESQview: A Guide to Programming the DESQview Multitasking Environment
Stephen R. Davis
Item #06-0 $39.95 (book/disk)
Item #28-1 $24.95 (book)
Fully endorsed by Quarterdeck Office Systems, publisher of DESQview, this book provides users with the information they need to get the most out of DESQview. Contents include the object-oriented DESQview 2.0 API (Application Program Interface) and multitasking concepts necessary to program the DESQview environment, example programs that control and interact with DESQview's API, and demonstrations of such concepts as windowing, intertask communication, and subtask control.

Interfacing to S-100/IEEE 696 Microcomputers
Mark Garetz and Sol Libes
Item #85-2 $24.95
This book helps S-100 bus users expand the utility and power of their systems. It describes the S-100 bus with unmatched precision. Various chapters describe its mechanical and functional design, logical and electrical relationships, bus interconnections, and busing techniques.

Building Local Area Networks
Patrick H. Corrigan and Scott Herndon
Item #025-7 $39.95 (book/disk)
Item #010-9 $24.95 (book)
The specifics of building and maintaining PC LANs, including hardware configurations, software development, cabling, selection criteria, installation, and on-going management, are described in a detailed, "how-to" manner with numerous illustrations and sample LAN management forms.

Dr. Dobb's Journal Bound Volume Series

Each volume in this series contains a full year's worth of useful code and fascinating history from *Dr. Dobb's Journal of Software Tools*. Each volume contains every issue of *DDJ* for a given year, reprinted and combined into one comprehensive reference.

Volume	1: 1976	*Item #13-5*	*$30.75*
Volume	2: 1977	*Item #16-X*	*$30.75*
Volume	3: 1978	*Item #17-8*	*$30.75*
Volume	4: 1979	*Item #14-3*	*$30.75*
Volume	5: 1980	*Item #18-6*	*$30.75*
Volume	6: 1981	*Item #19-4*	*$30.75*
Volume	7: 1982	*Item #20-8*	*$35.75*
Volume	8: 1983	*Item #00-3*	*$35.75*
Volume	9: 1984	*Item #08-9*	*$35.75*
Volume	10: 1985	*Item #21-6*	*$35.75*
Volume	11: 1986	*Item #72-0*	*$35.75*
Volume	12: 1987	*Item #84-4*	*$39.95*
Volume	13: 1988	*Item #27-3*	*$39.75*

To order any of these products send your payment, along with $2.95 per item for shipping, to M&T Books, 501 Galveston Drive, Redwood City, California 94063. California residents, please include the appropriate sales tax. Or, call toll-free 800-533-4372 (in California 800-356-2002) Monday through Friday between 8 A.M. and 5 P.M. Pacific Standard Time. When ordering disks, please indicate format.

The Dr. Dobb's Bound Volume Series

Volume 1—1976

Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, speech synthesis, floating point routines, the 6502 disassembler for the Apple, and much more.

Volume 2—1977

The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starkweather's PILOT, using a modem, string-handling techniques, ciphers, Turtle graphics, and micro utilities.

Volume 3—1978

Explores the foundation of the mass-market computer industry in 1978. Topics include: programming in BASIC, PILOT, and Pascal; RAM memory testers; Apple utility programs; the S-100 bus standard; STRUBAL; and a structured BASIC compiler.

Volume 4—1979

Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: selecting business software, microcomputer speech and music, information networks, and interfacing techniques.

Volume 5—1980

Focuses on the technological promise of the modern microcomputer and the creative challenge facing programmers in 1980. Topics include: the revolutionary impact of CP/M, C programming and the UNIX operating systems, a survey of computer networks, software portability, introduction to Forth, and compiler writing.

Volume 6—1981

The microcomputer enters the mainstream—these issues assess the revolutionary impact of microcomputers on the individual and on society in 1981. Topics include: computer conferencing, the power of Forth, 8- and 16-bit technology, Rubik's cube simulator, and an adventure game development system.

Volume 7—1982

Examines the potential of powerful 16-bit micros, including the birth of the IBM PC in 1982. Topics include: in-depth coverage of Forth, 68000, 8088 programming, C software tools, and utilities for CP/M—including a spelling checker, using bulletin boards, and more.

Volume 8—1983

DDJ turns pro. Some of the most powerful professional programmer's tools ever published in a magazine are in this 1983 volume, including Small C, the RED editor, and an Ada subset.

Volume 9—1984

Shaping things to come. In 1984 *DDJ* examined new programming environments, Prolog, expert systems, Modula-2, and Pascal. Other topics include GREP, UNIX internals, and two encryptions systems.

Volume 10—1985

The year of living dangerously. In 1985 *DDJ* added more memory, an SCSI port, and a hard disk to the Macintosh; challenged the UNIX establishment with plans for a free UNIX; and asked hard questions about privacy and control in the Information Age. Includes software tools in C, Modula-2, Forth, Pascal, assembly language, and Prolog.

Volume 11—1986

The promise of power. Desktop computers began to rival minis and mainframes in power. *DDJ* covered 1986's changes with issues on the 68000, parallel processing, artificial intelligence, the 80386, and multitasking. *DDJ* also supported the new chips with assemblers, translators, and other development tools.

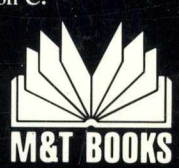
Volume 12—1987

Building better brains. *DDJ* broke new ground in 1987 with a neural network implementation, monthly coverage of artificial intelligence and object-oriented programming techniques, and reviews of implementations of Prolog, LISP, and Smalltalk.

Volume 13—1988

The new frontier. Database technology, Hyperanimation, and much more! *DDJ* talked about how to create a new language, write real-time programs under UNIX, and develop good software. Other topics included object-oriented programming, SQL development tools, and more on C.

ISBN 1-55851-027-3
>$39.95



M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063